US006243102B1

(12) **United States Patent**
**Ruff et al.**

(10) **Patent No.: US 6,243,102 B1**
(45) **Date of Patent: Jun. 5, 2001**

(54) **DATA-DRIVEN LAYOUT ENGINE**

(75) Inventors: **Joseph Ruff**, Palo Alto, CA (US); **Robert G. Johnston, Jr.**, Gainesville, FL (US); **Robert Ulrich**, Mountain View, CA (US)

(73) Assignee: **Apple Computer, Inc.**, Cupertino, CA (US)

( * ) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 0 days.

(21) Appl. No.: **09/425,738**

(22) Filed: **Oct. 22, 1999**

**Related U.S. Application Data**

(63) Continuation of application No. 08/644,360, filed on May 10, 1996, and a continuation of application No. 08/242,963, filed on May 16, 1994, now abandoned, and a continuation-in-part of application No. 08/243,368, filed on May 16, 1994, and a continuation-in-part of application No. 08/243,327, filed on May 16, 1994.

(51) **Int. Cl.**[7] ..... **G06F 15/00**
(52) **U.S. Cl.** ..... **345/433**
(58) **Field of Search** ..... 345/441, 435, 345/433, 440, 418, 439, 426; 707/502, 517, 520

(56) **References Cited**

U.S. PATENT DOCUMENTS

| | | | | |
|---|---|---|---|---|
| 6,011,559 | * | 9/2000 | Gangopadhyay | 345/435 |
| 6,067,070 | * | 5/2000 | Suzuki et al. | 345/439 |
| 6,118,897 | * | 9/2000 | Kohno | 345/440 |

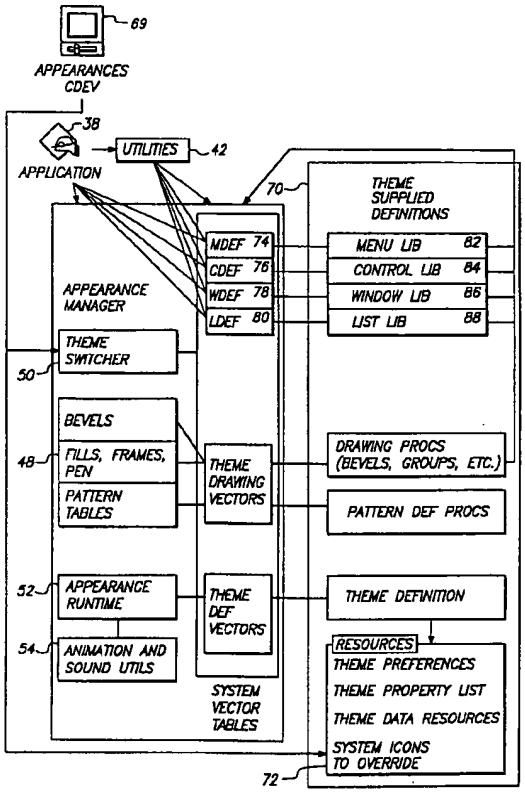
* cited by examiner

*Primary Examiner*—Phu K. Nguyen
(74) *Attorney, Agent, or Firm*—Burns, Doane, Swecker & Mathis, L.L.P.

(57) **ABSTRACT**

Systems and methods for providing a user with increased flexibility and control over the appearance and behavior of objects on a user interface are described. Sets of objects can be grouped into themes to provide a user with a distinct overall impression of the interface. These themes can be switched dynamically by switching pointers to drawing procedures or switching data being supplied to these procedures. To buffer applications from the switchable nature of graphical user interfaces according to the present invention, colors and patterns used to implement the interface objects are abstracted from the interface by, for example, pattern look-up tables.

**32 Claims, 16 Drawing Sheets**



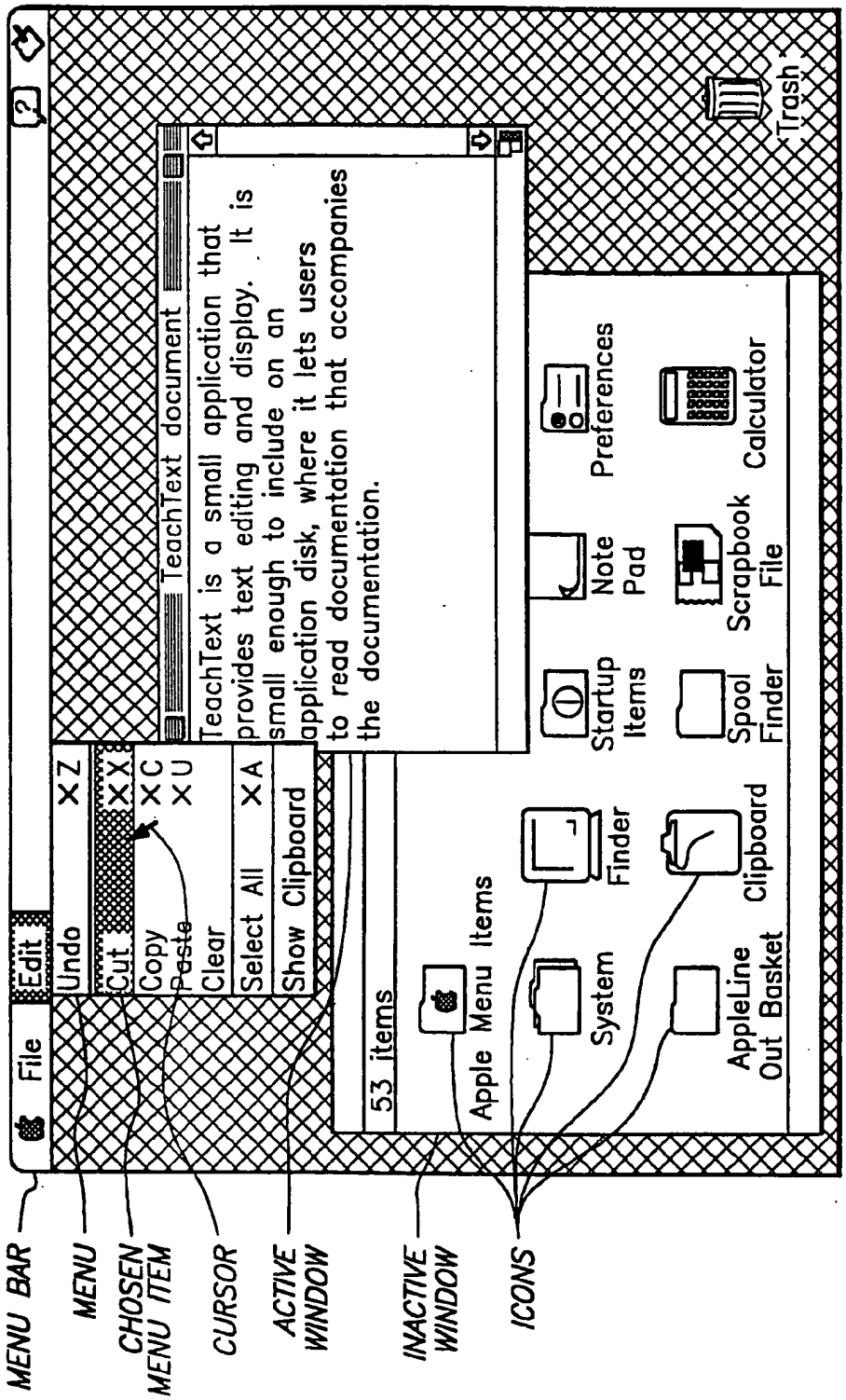

MENU BAR
MENU
CHOSEN MENU ITEM
CURSOR
ACTIVE WINDOW
INACTIVE WINDOW
ICONS
File
Edit
Undo ⌘Z
Cut ⌘X
Copy ⌘C
Paste ⌘U
Clear
Select All ⌘A
Show Clipboard
TeachText document
TeachText is a small application that provides text editing and display. It is small enough to include on an application disk, where it lets users to read documentation that accompanies the documentation.
53 items
Apple Menu Items
System Finder
AppleLine Out Basket
Clipboard
Startup Items
Spool Finder
Note Pad
Scrapbook File
Preferences
Calculator
Trash

FIG. 1

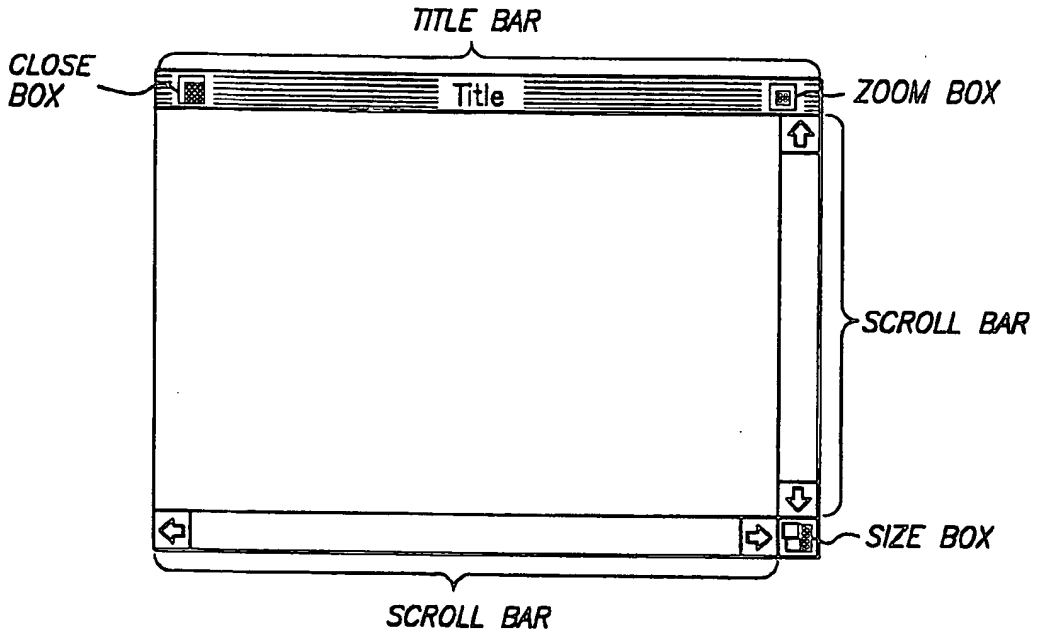

TITLE BAR
CLOSE BOX
Title
ZOOM BOX
SCROLL BAR
SIZE BOX
SCROLL BAR

FIG. 2A

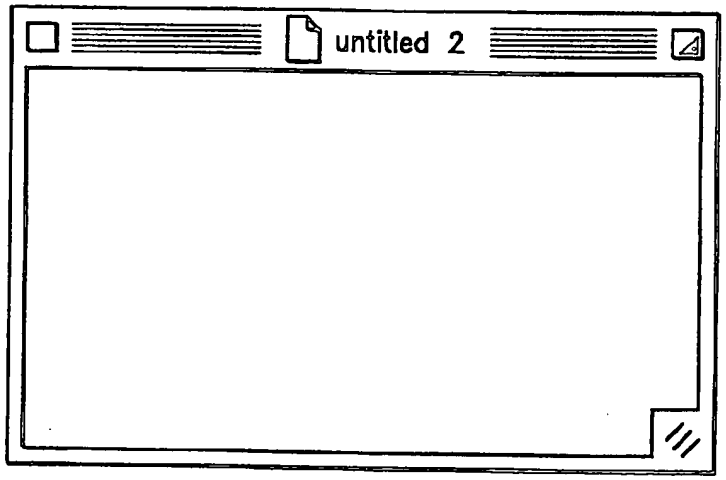

untitled 2

FIG. 2B

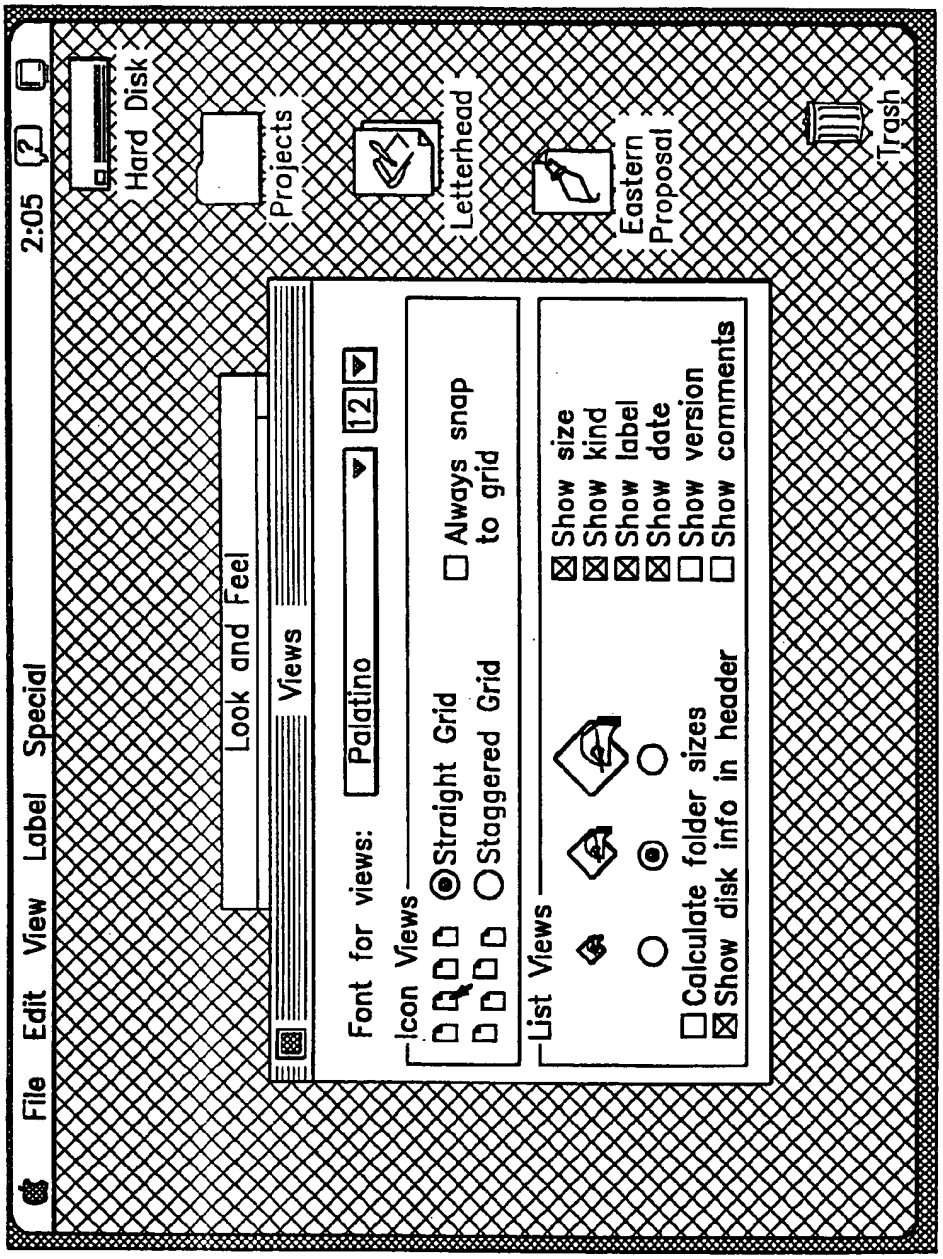

File Edit View Label Special
2:05
Hard Disk
Projects
Letterhead
Eastern Proposal
Trash
Look and Feel
Views
Font for views:
Palatino
12
Icon Views
Straight Grid
Staggered Grid
Always snap to grid
List Views
Show size
Show kind
Show label
Show date
Show version
Show comments
Calculate folder sizes
Show disk info in header

FIG. 2C

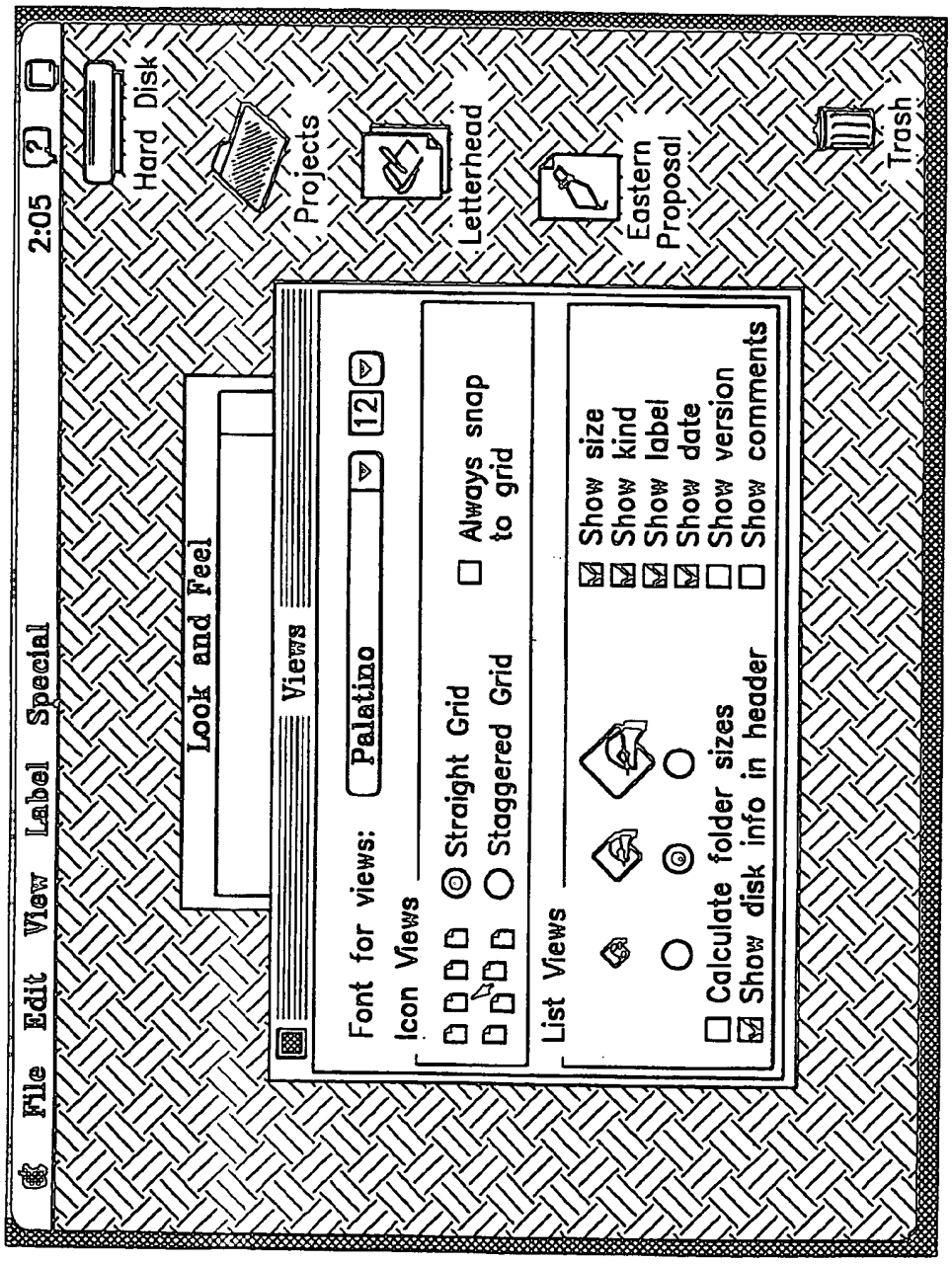

File Edit View Label Special
2:05
Hard Disk
Projects
Letterhead
Eastern Proposal
Trash
Look and Feel
Views
Font for views:
Palatino
12
Icon Views
Straight Grid
Staggered Grid
Always snap to grid
List Views
Show size
Show kind
Show label
Show date
Show version
Show comments
Calculate folder sizes
Show disk info in header

FIG. 2D

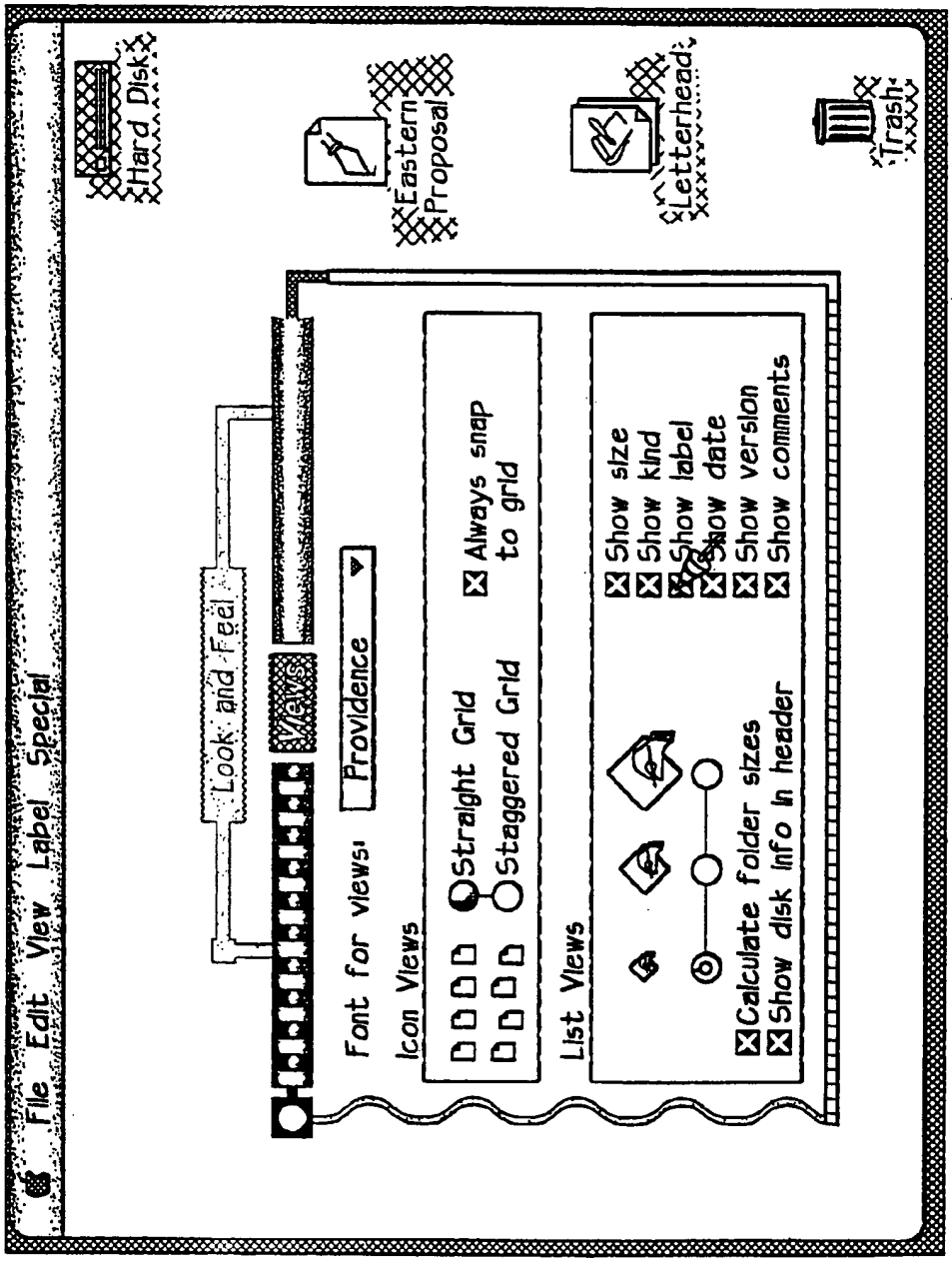

File Edit View Label Special
Hard Disk
Eastern Proposal
Letterhead
Trash
Look and Feel
Views
Font for views:
Providence
Icon Views
Straight Grid
Staggered Grid
Always snap to grid
List Views
Show size
Show kind
Show label
Show date
Show version
Show comments
Calculate folder sizes
Show disk info in header

FIG. 2E

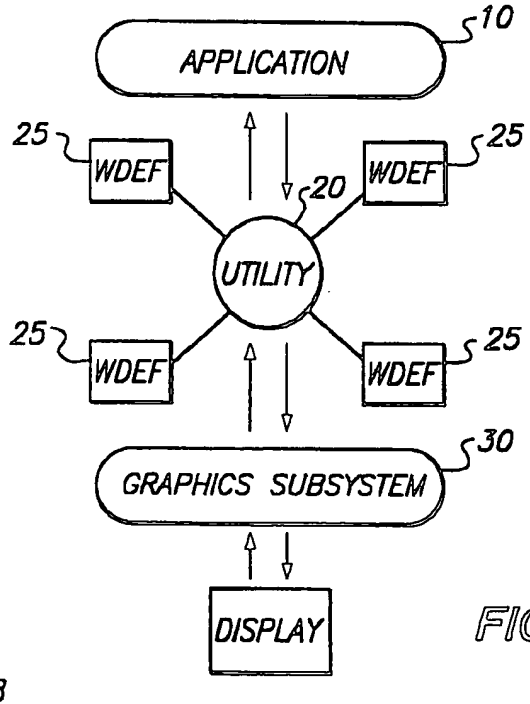

APPLICATION
10
25
WDEF
25
WDEF
20
UTILITY
25
WDEF
25
WDEF
30
GRAPHICS SUBSYSTEM
DISPLAY

FIG. 3

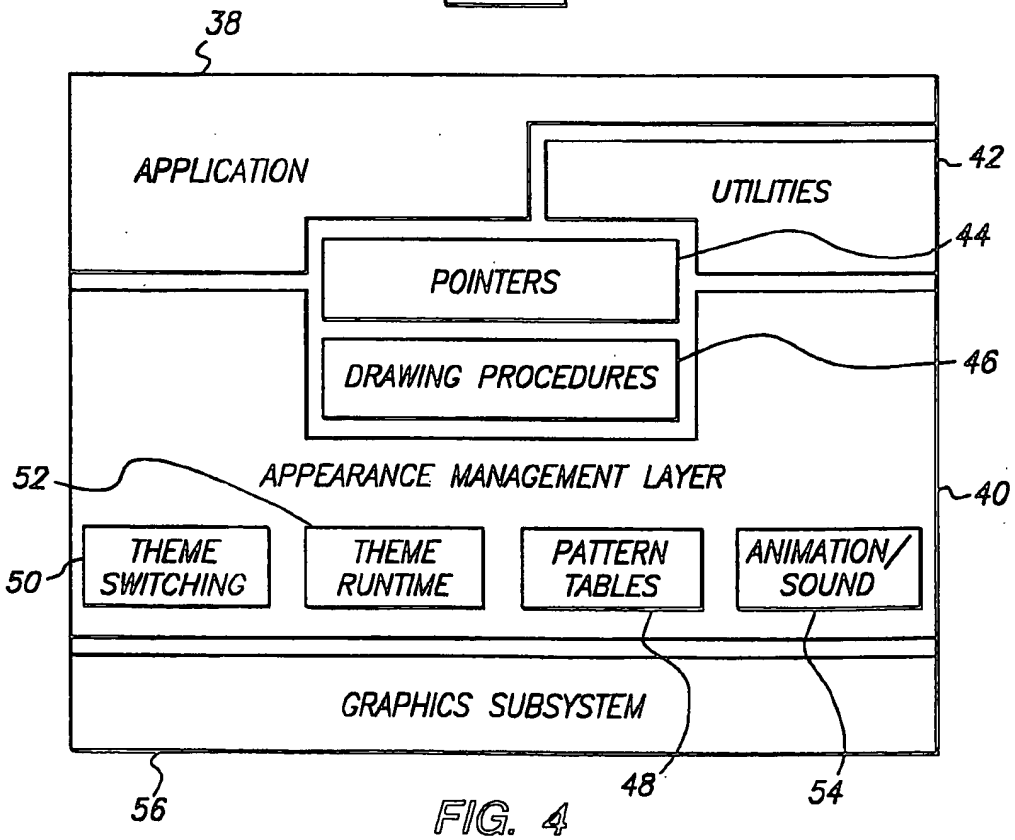

38
APPLICATION
UTILITIES
42
44
POINTERS
DRAWING PROCEDURES
46
52
APPEARANCE MANAGEMENT LAYER
40
50
THEME SWITCHING
THEME RUNTIME
PATTERN TABLES
ANIMATION/ SOUND
GRAPHICS SUBSYSTEM
48
54
56

FIG. 4

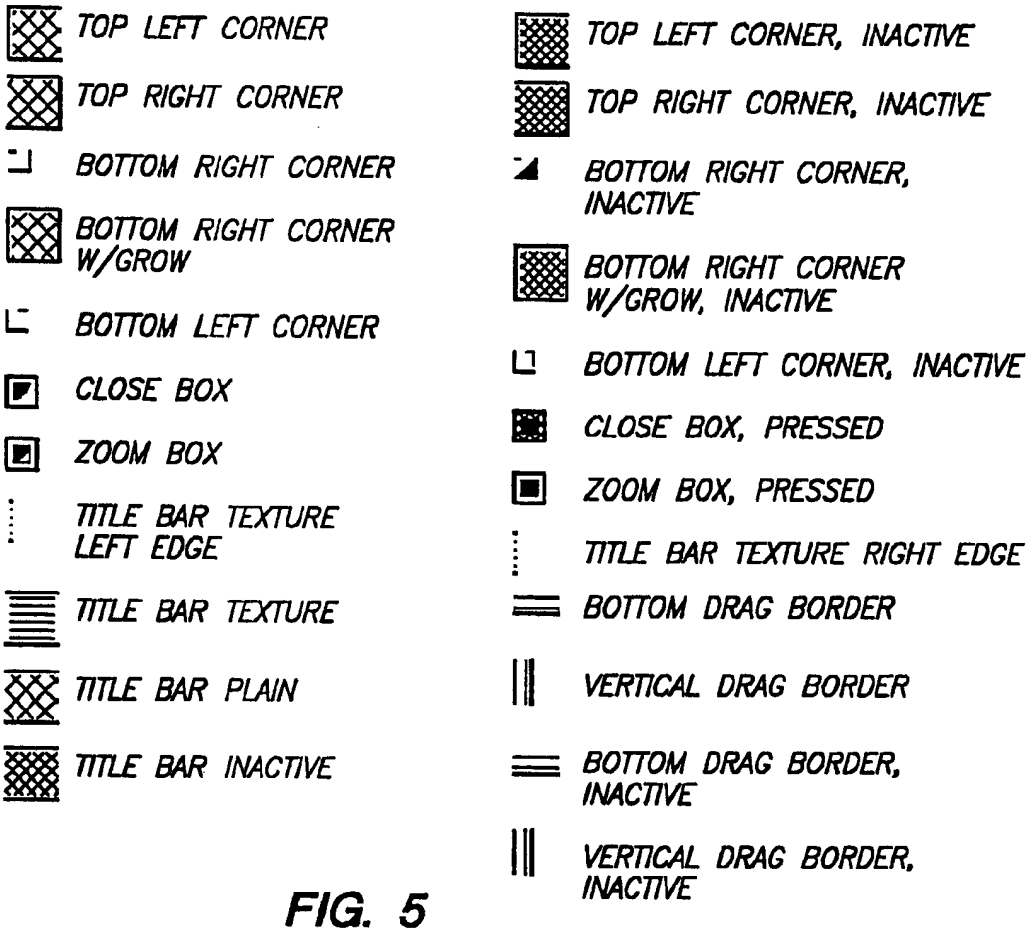



FIG. 5

TO

| FROM | Q1 | Q2 | Q3 | Q4 | Q5 | Q6 | Q7 | Q8 | Q9 |
|---|---|---|---|---|---|---|---|---|---|
| Q1 | | T1 | T3 | T5 | | | T7 | | |
| Q2 | T2 | | | | | | | | |
| Q3 | T4 | | | | | | | | |
| Q4 | T6 | | | | T9 | T11 | T13 | | |
| Q5 | | | | T10 | | | | | |
| Q6 | | | | T12 | | | | | |
| Q7 | T8 | | | T14 | | | | T15 | T17 |
| Q8 | | | | | | | T16 | | |
| Q9 | | | | | | | T18 | | |

FIG. 7

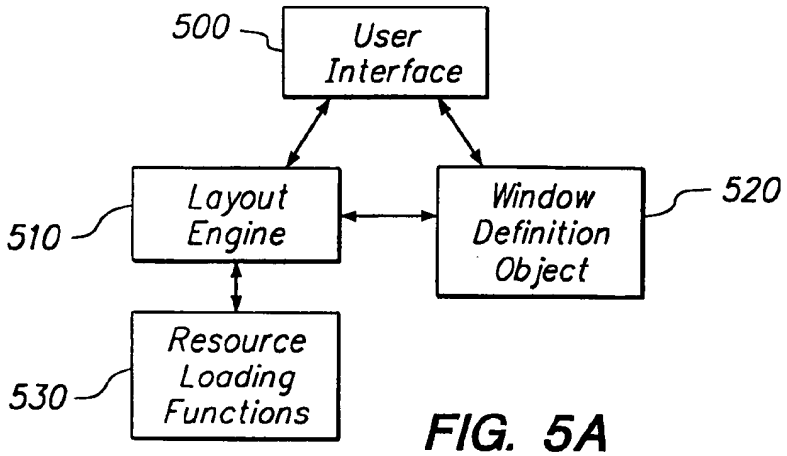



FIG. 5A

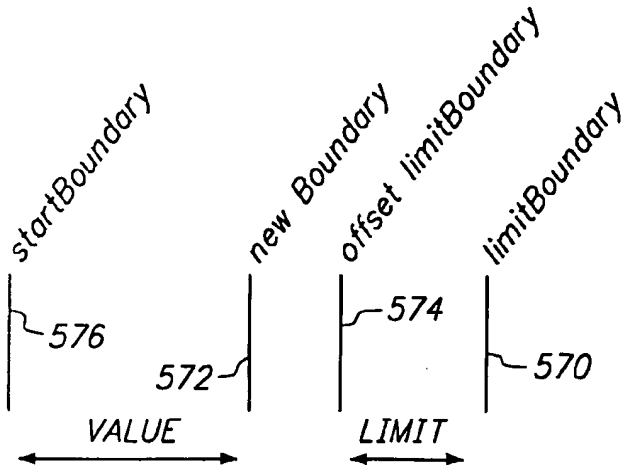



FIG. 5B

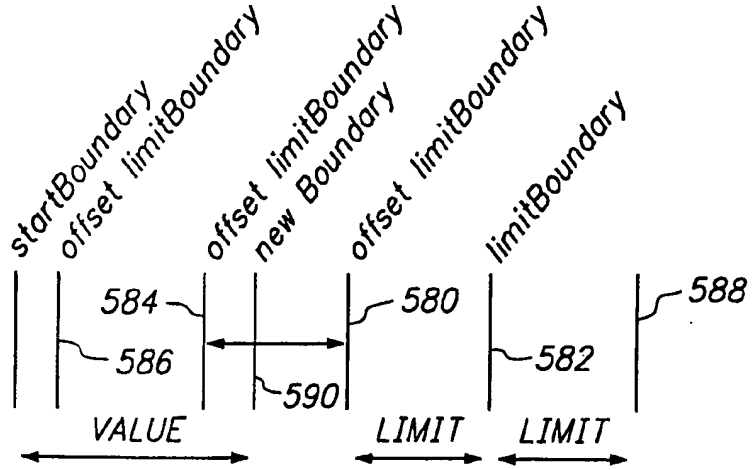



FIG. 5C

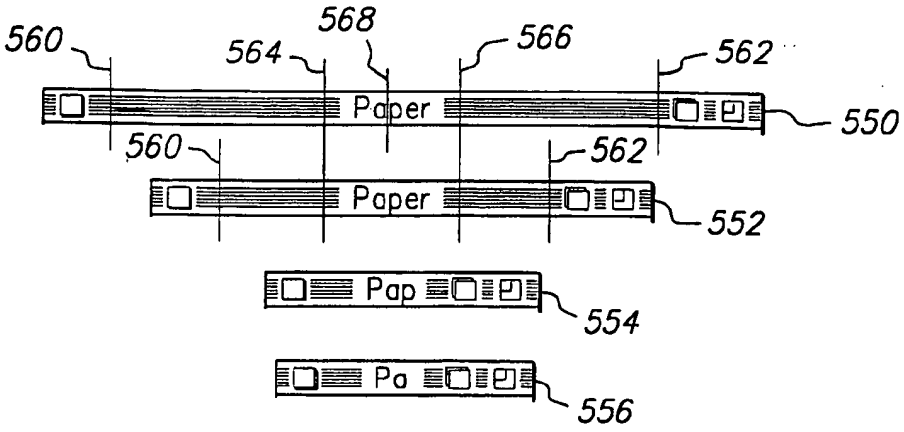

560
564
568
566
562
Paper
550
560
562
Paper
552
Pap
554
Pa
556

FIG. 5D

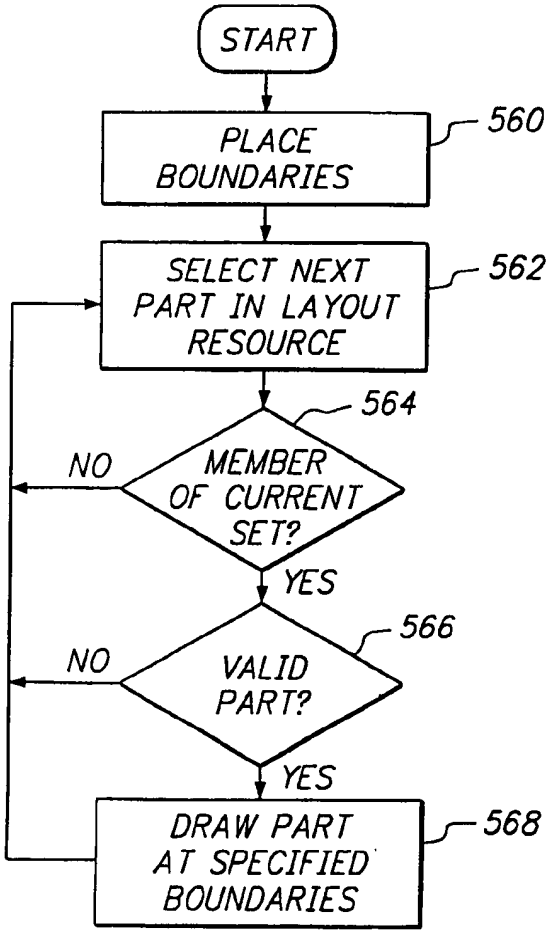

START
PLACE BOUNDARIES
560
SELECT NEXT PART IN LAYOUT RESOURCE
562
564
MEMBER OF CURRENT SET?
NO
YES
566
VALID PART?
NO
YES
DRAW PART AT SPECIFIED BOUNDARIES
568

FIG. 5E

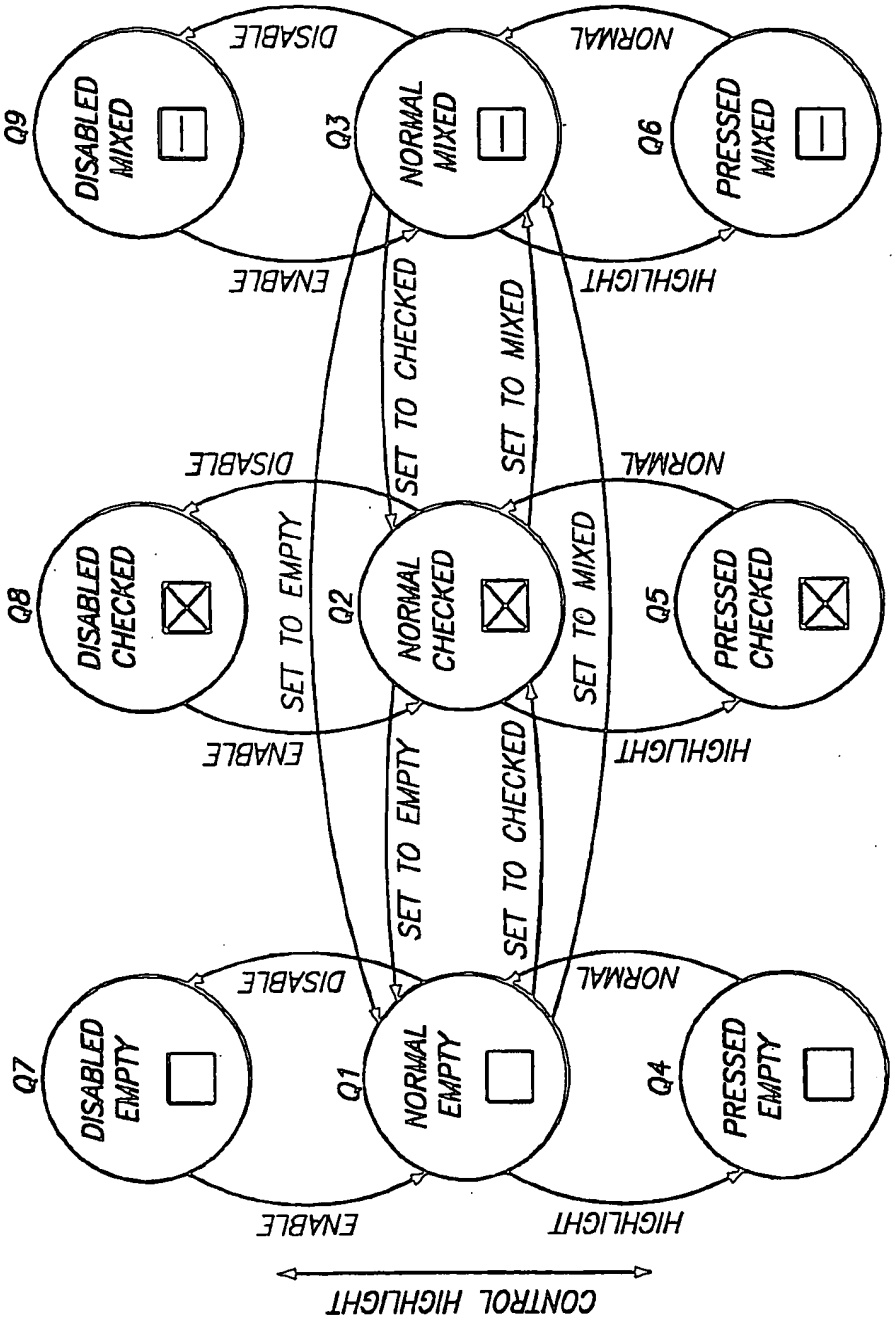

Q9
DISABLED MIXED
Q3
NORMAL MIXED
Q6
PRESSED MIXED
DISABLE
NORMAL
ENABLE
HIGHLIGHT
SET TO CHECKED
SET TO MIXED
Q8
DISABLED CHECKED
Q2
NORMAL CHECKED
Q5
PRESSED CHECKED
SET TO EMPTY
SET TO MIXED
Q7
DISABLED EMPTY
Q1
NORMAL EMPTY
Q4
PRESSED EMPTY
SET TO EMPTY
SET TO CHECKED
CONTROL VALUE
CONTROL HIGHLIGHT

FIG. 6

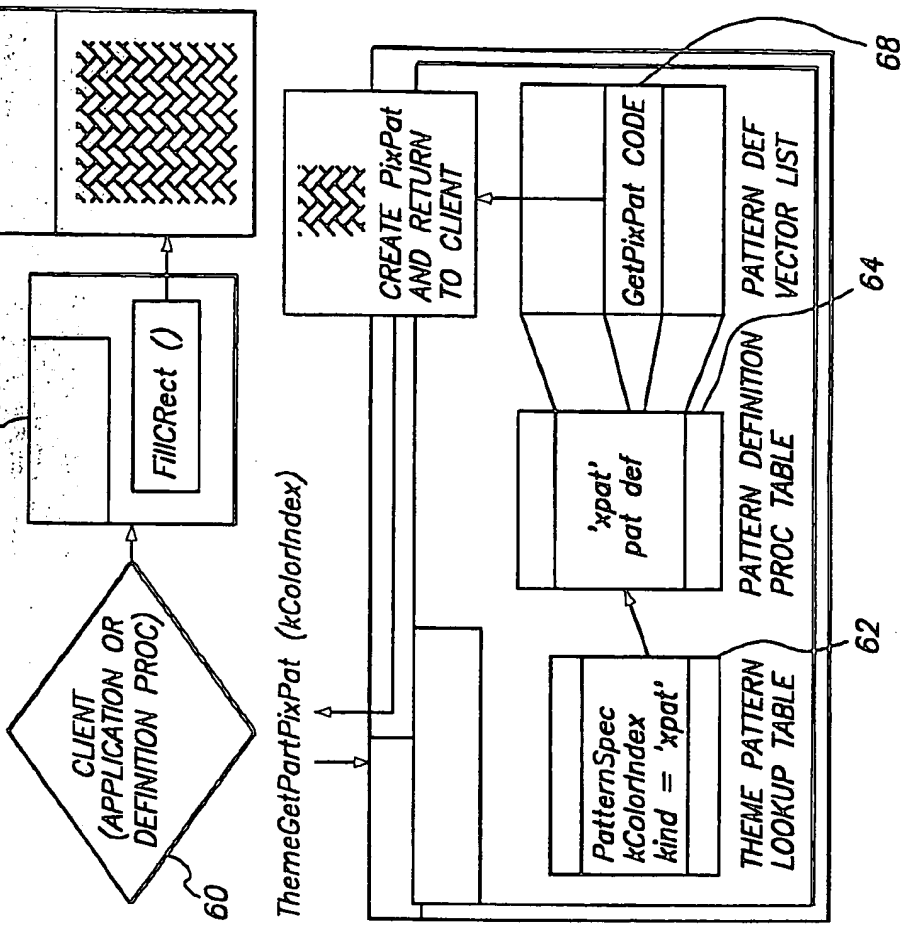

CLIENT (APPLICATION OR DEFINITION PROC)
60
56
FillCRect ()
ThemeGetPartPixPat (kColorIndex)
CREATE PixPat AND RETURN TO CLIENT
GetPixPat CODE
'xpat' pat def
PatternSpec kColorIndex kind = 'xpat'
PATTERN DEF VECTOR LIST
PATTERN DEFINITION PROC TABLE
THEME PATTERN LOOKUP TABLE
64
62
68

FIG. 10

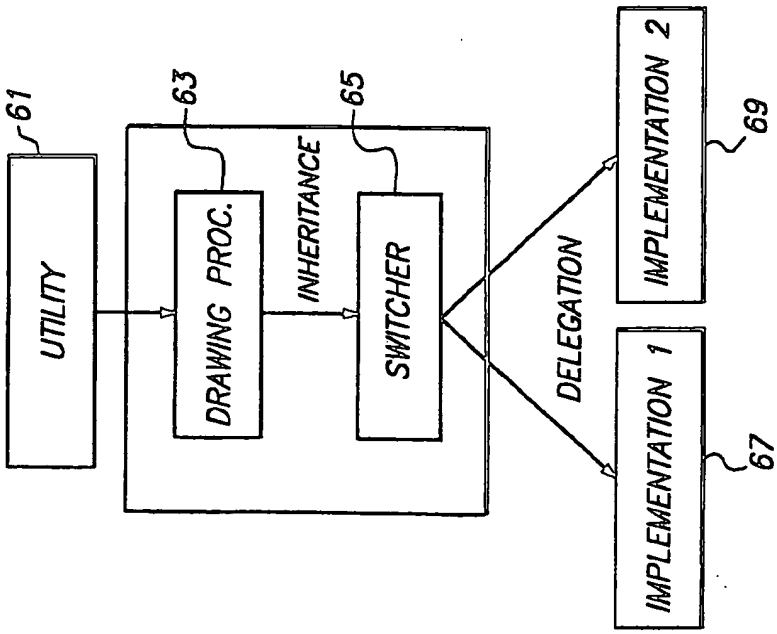

UTILITY
61
DRAWING PROC.
63
INHERITANCE
SWITCHER
65
DELEGATION
IMPLEMENTATION 1
67
IMPLEMENTATION 2
69

FIG. 8

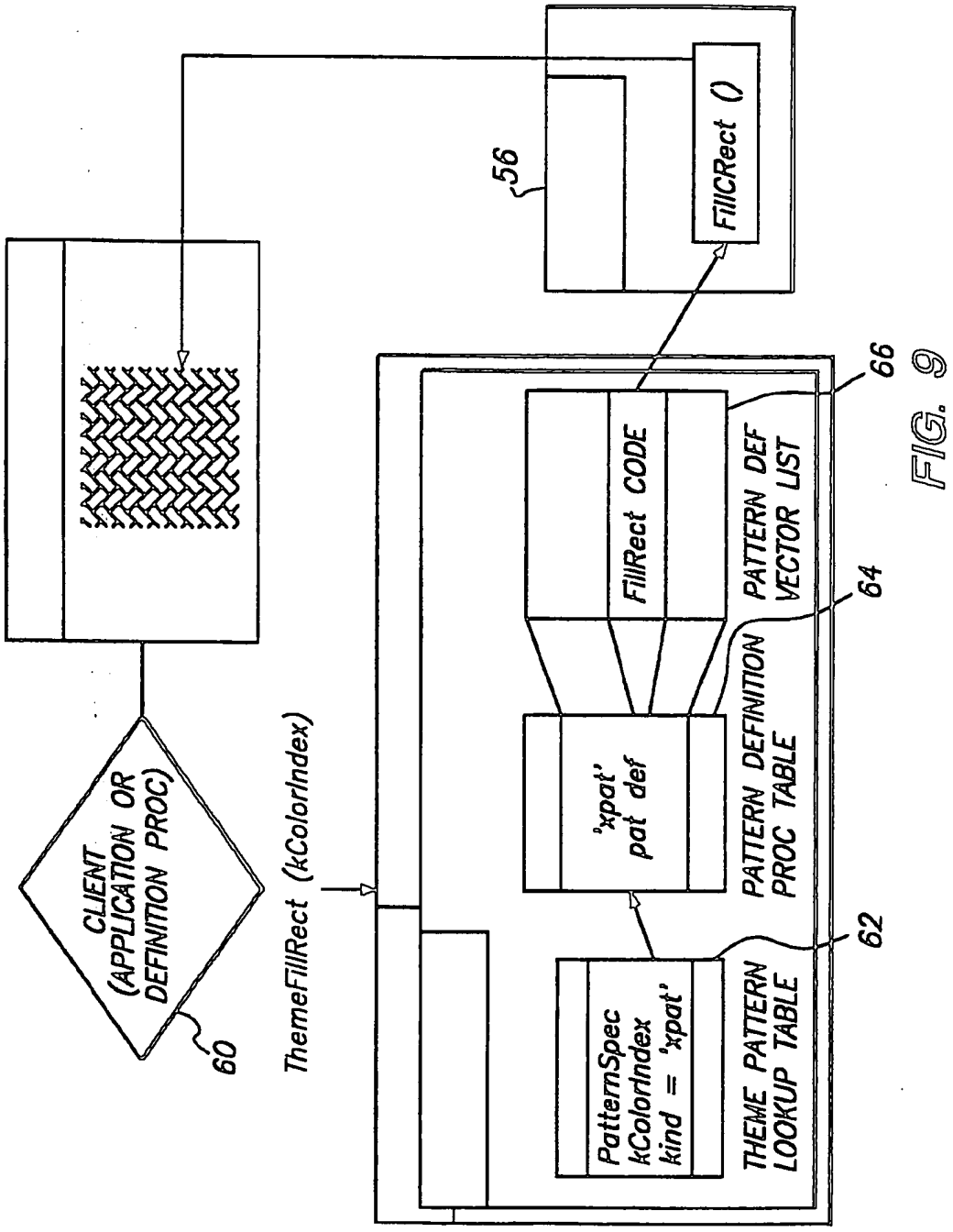

CLIENT (APPLICATION OR DEFINITION PROC)
60
ThemeFillRect (kColorIndex)
56
FillCRect ()
66
FillRect CODE
PATTERN DEF VECTOR LIST
64
'xpat' pat def
PATTERN DEFINITION PROC TABLE
62
PatternSpec kColorIndex kind = 'xpat'
THEME PATTERN LOOKUP TABLE

FIG. 9

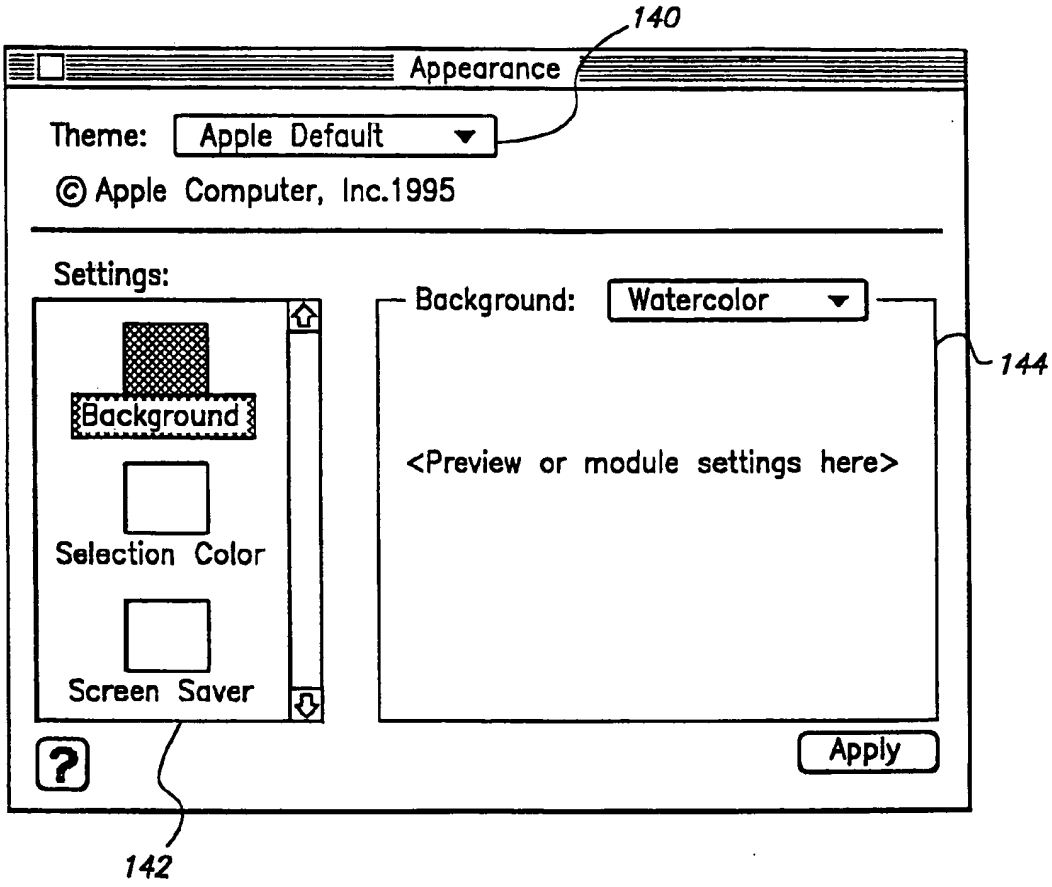

140
Appearance
Theme:
Apple Default
© Apple Computer, Inc.1995
Settings:
Background:
Watercolor
144
Background
Selection Color
Screen Saver
<Preview or module settings here>
Apply
142

FIG. 11

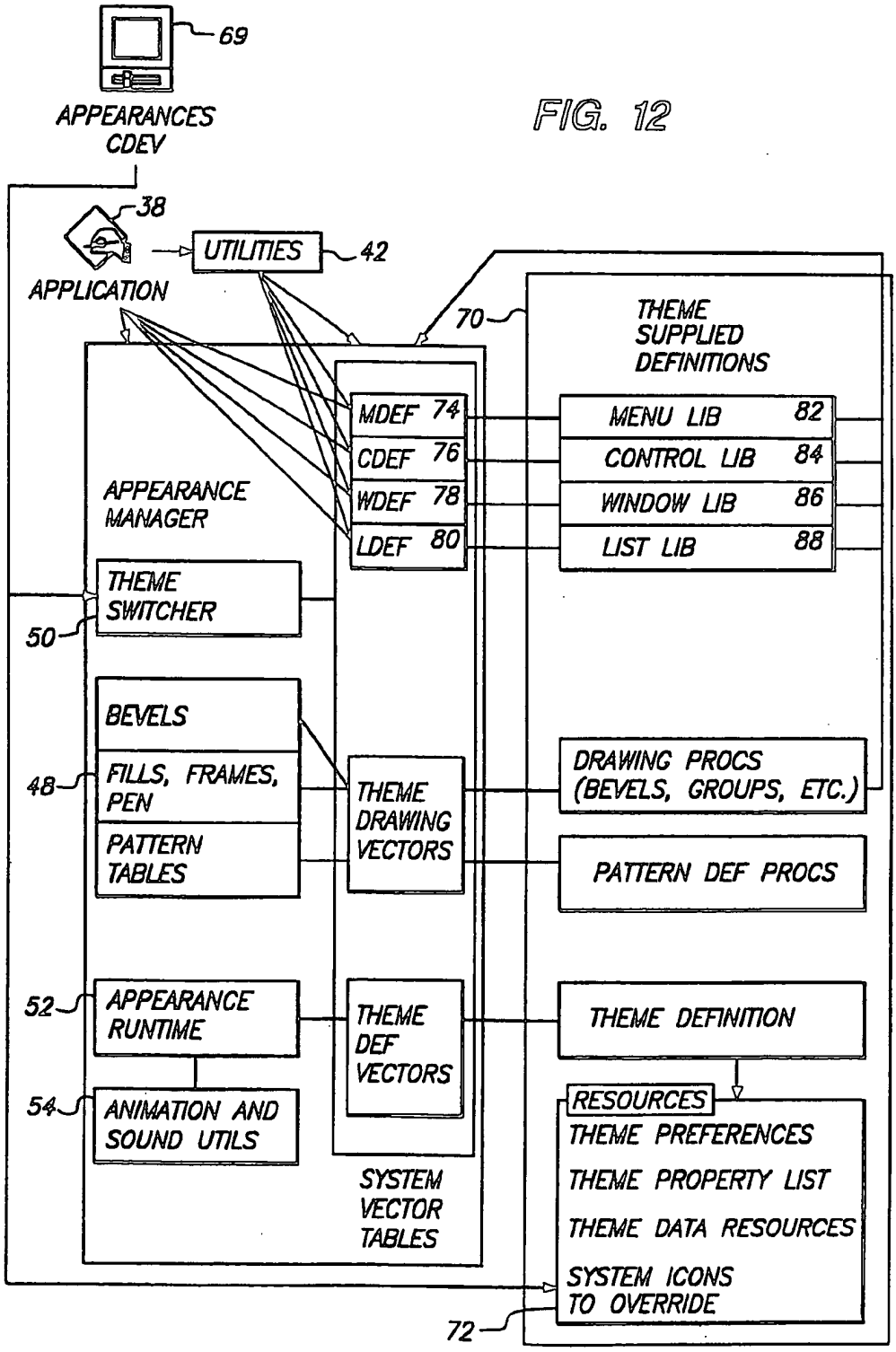

69
APPEARANCES CDEV
38
APPLICATION
UTILITIES
42
70
THEME SUPPLIED DEFINITIONS
MDEF 74
CDEF 76
WDEF 78
LDEF 80
MENU LIB 82
CONTROL LIB 84
WINDOW LIB 86
LIST LIB 88
APPEARANCE MANAGER
THEME SWITCHER
50
BEVELS
FILLS, FRAMES, PEN
PATTERN TABLES
48
THEME DRAWING VECTORS
DRAWING PROCS (BEVELS, GROUPS, ETC.)
PATTERN DEF PROCS
52
APPEARANCE RUNTIME
54
ANIMATION AND SOUND UTILS
THEME DEF VECTORS
THEME DEFINITION
RESOURCES
THEME PREFERENCES
THEME PROPERTY LIST
THEME DATA RESOURCES
SYSTEM ICONS TO OVERRIDE
SYSTEM VECTOR TABLES
72

FIG. 12

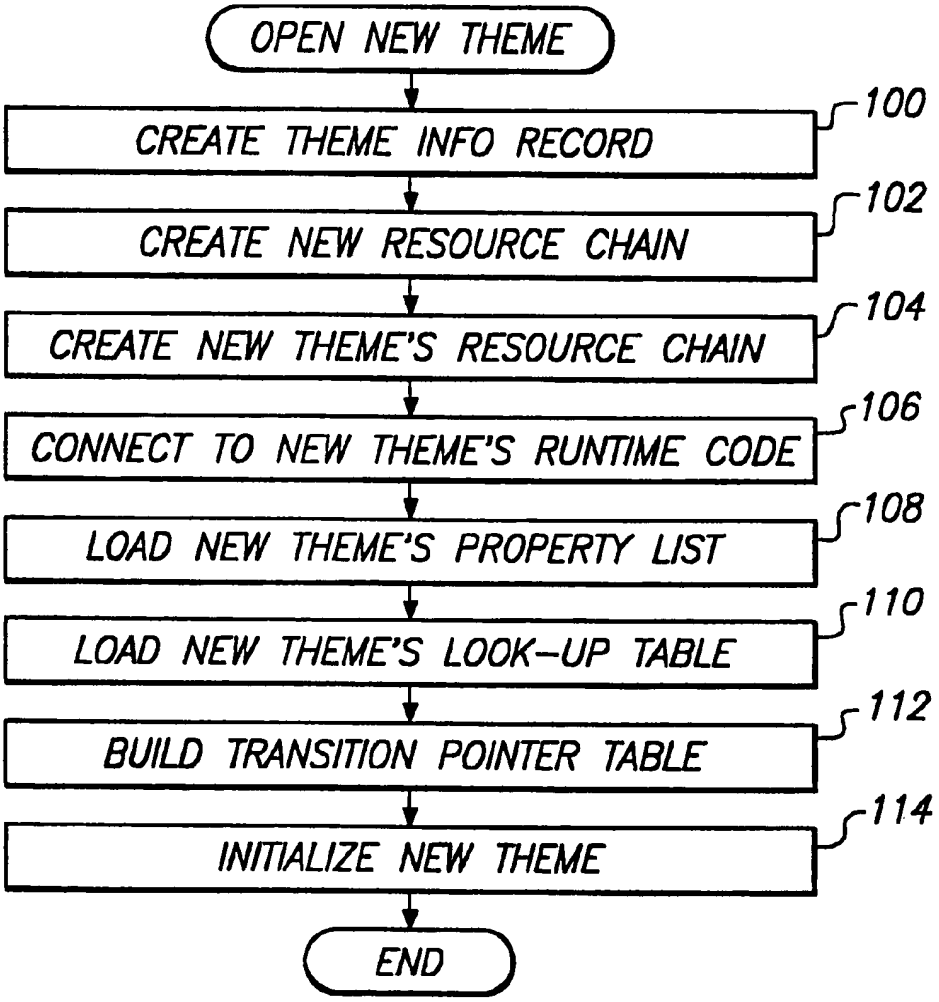

OPEN NEW THEME
CREATE THEME INFO RECORD
100
CREATE NEW RESOURCE CHAIN
102
CREATE NEW THEME'S RESOURCE CHAIN
104
CONNECT TO NEW THEME'S RUNTIME CODE
106
LOAD NEW THEME'S PROPERTY LIST
108
LOAD NEW THEME'S LOOK-UP TABLE
110
BUILD TRANSITION POINTER TABLE
112
INITIALIZE NEW THEME
114
END

FIG. 13

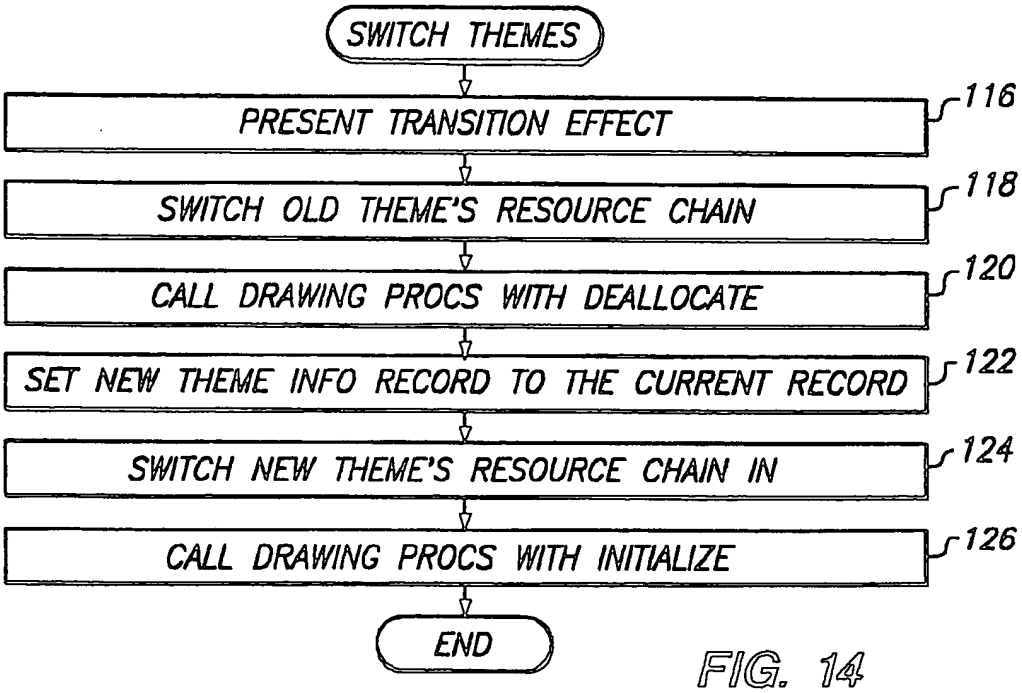

SWITCH THEMES
PRESENT TRANSITION EFFECT
116
SWITCH OLD THEME'S RESOURCE CHAIN
118
CALL DRAWING PROCS WITH DEALLOCATE
120
SET NEW THEME INFO RECORD TO THE CURRENT RECORD
122
SWITCH NEW THEME'S RESOURCE CHAIN IN
124
CALL DRAWING PROCS WITH INITIALIZE
126
END

FIG. 14

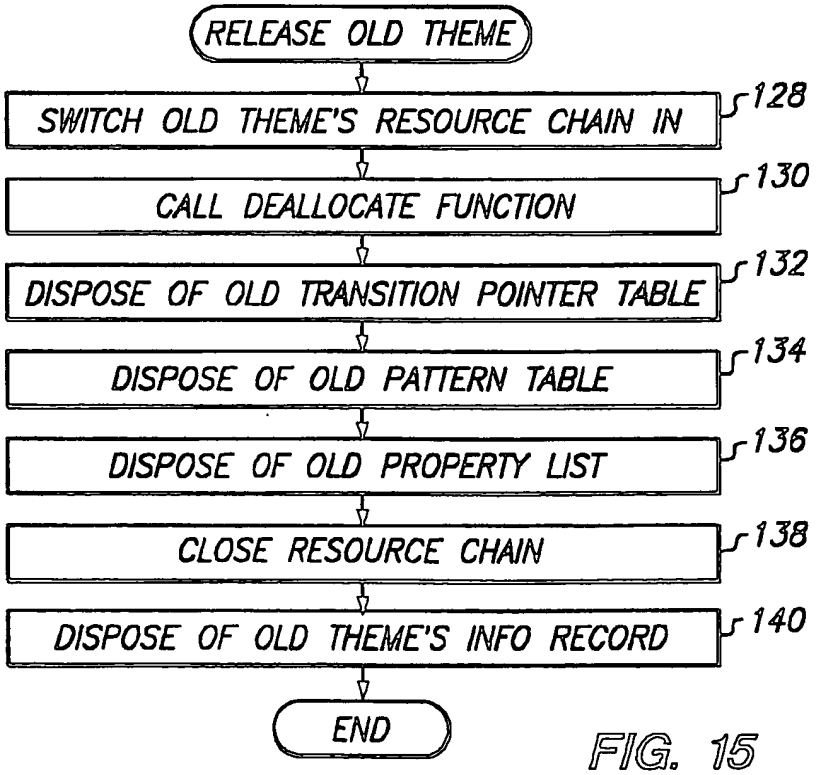

RELEASE OLD THEME
SWITCH OLD THEME'S RESOURCE CHAIN IN
128
CALL DEALLOCATE FUNCTION
130
DISPOSE OF OLD TRANSITION POINTER TABLE
132
DISPOSE OF OLD PATTERN TABLE
134
DISPOSE OF OLD PROPERTY LIST
136
CLOSE RESOURCE CHAIN
138
DISPOSE OF OLD THEME'S INFO RECORD
140
END

FIG. 15

# DATA-DRIVEN LAYOUT ENGINE

## RELATED APPLICATIONS

This application is a continuation, of Application Ser. No. 08/644,360, filed May 10, 1996 and a continuation-in-part of U.S. patent application Ser. No. 08/242,963 abandoned entitled "Pattern and Color Abstraction in a Graphical User Interface", U.S. patent application Ser. No. 08/243,368 entitled "Switching Between Appearance/Behavior Themes in Graphical User Interfaces" and U.S. patent application Ser. No. 08/243,327 entitled "A System and Method for Customizing Appearance and Behavior of Graphical User Interfaces", all of which were filed on May 16, 1994 and all of which are hereby incorporated by reference.

## COPYRIGHT NOTICE



## BACKGROUND

The present invention relates generally to graphical user interfaces for computer systems. More particularly, the present invention relates to systems and methods for interfacing applications and operating systems which provide for flexible customization of graphical user interfaces.

The evolution of the computer industry is unparalleled in its rate of growth and complexity. Personal computers, for example, which began as little more than feeble calculators with limited memory, tape-driven input and monochrome displays are now able to tackle almost any data processing task. While this meteoric increase in power was almost sufficient to satisfy the demand of application programmers and end users alike, the corresponding increase in complexity created an ease-of-use problem which the industry was somewhat slower in solving. Thus, designers were faced with a new challenge: to harness this computing power in a form usable by even those with relatively little computer training to smooth the transition of other industries into a computer-based information paradigm.

As a result, in the early to mid-1980's many new I/O philosophies, such as "user friendly", "WYSIWYG" and "menu driven" came to the forefront of the industry. These concepts are particularly applicable to microcomputers, also known as personal computers, which are intended to appeal to a broad audience of computer users, including those who previously feared and mistrusted computers. An important aspect of computers which employ these concepts was, and continues to be, the interface which allows the user to input commands and data and receive results, which is commonly referred to as a graphical user interface (GUI).

One type of GUI display is based on a visual metaphor which uses a monitor screen as a work surface called a "desktop" where documents are presented in relocatable regions termed "windows". The user interacts with the computer by, for example, moving objects on the desktop, choosing commands from menus, and manipulating window controls, such as checkboxes and scroll bars. An exemplary desktop screen is reproduced as FIG. **1**.

The success of this type of interface is evident from the number of companies which have emulated the desktop environment. Even successful concepts, however, must continually be improved in order to keep pace with the rapid growth in this industry. The advent of multimedia, especially CD-ROM devices, has provided vast quantities of secondary storage which have been used to provide video capabilities, e.g., live animation and video clips, as regular components of application displays. With these new resources at their disposal, application designers, and others, desire more and more control over the appearance of the display, including the desktop environment and, in particular, objects on the desktop.



Windows are one example of desktop objects which can be virtually any size, shape, or color. Some standard types of windows are commonly predefined for the interface including, for example, a document window and a dialog box. One example of a standard for a document window is illustrated in FIG. **2**A. Each document window which conforms to this standard has a title bar with a title drawn in a system-defined font and color. Active document windows can also have controls as illustrated in FIG. **2**A, for example, a close box, a zoom box, a size box, and scroll bars. These standard types of windows (as well as other standard desktop objects) are beyond the reach of users who wish to alter the appearance and/or behavior.

Accordingly, application developers can define their own nonstandard window types as desired, although each nonstandard window requires a relatively large block of memory. Further, even these nonstandard window types provide only limited flexibility and control over the appearance and behavior of desktop objects in that they are application-specific and do not present a consistent interface across all applications, i.e., if three different applications are running, each might present a different "look" on desktop. Once again, the user has virtually no control over the appearance and/or behavior of these nonstandard window objects.

Since the window format, including the appearance, behavior and function of standard windows and window parts, is known a priori to applications which were designed for such conventional systems, these applications are written to take advantage of such knowledge. As seen in FIG. **3**, suppose, for example, that an application **10** desires to draw a rectangle in the color of the title bar (beige, in this example) in a window (not shown on the desktop). The application assumes knowledge of the color of the title bar when using predefined standard window definitions **25** and, if this application uses window definitions created by the application itself, the application will have actual knowledge of colors defined by those windows. Accordingly, the application will simply send a command to the interface instructing that a beige rectangle be drawn in the window.

Each standard window, as well as any nonstandard window, conventionally has a corresponding window definition **25**. The window definition **25** includes all of the data necessary to define the window. Looking at the active window illustrated in FIG. **1**, data included in the window definition **25** for such an active window would include, for example, the size of the window, the relative location of the close box and zoom box in the upper lefthand and righthand corners, respectively, the number of parallel lines and their locations relative to the close box and the zoom box, and the upper boundary of the window and all of the other defining features of that particular window. The application supplies the variable parameters such as the location of the window on the desktop interface and, perhaps, the colors and/or fonts to be used for the text and/or figures in the window. As one can imagine, the window definitions can include a large amount of data and, therefore, can require a large amount of memory for each definition.

In addition to the amount of memory used to create non-standard window definitions, another problem with this conventional method of providing variety of appearance in the graphical user interface is the lack of a consistent appearance between objects drawn on the desktop by different applications. With multitasking i.e., multiple applications running simultaneously on a desktop, it is now common for users to simultaneously run multiple applications each of which has its own window on the desktop. However, if each application uses its own combination of standard and non-standard window definitions that result in each application having its own appearance and behavior. The dissimilarity in appearance and behavior between applications can be annoying and confusing to a user.

Accordingly, it would be desirable to allow application designers and application users to have additional flexibility and greater control over the appearance and behavior of desktop objects and individual controls for those objects.

## SUMMARY

According to exemplary embodiments of the present invention, an improved visual appearance can be provided to GUIs by providing a layout engine which provides a data-driven facility to customize the appearance and behavior of the desktop. This layout engine can, for example, be designed to receive commands from definition objects and provide instructions to the graphic subsystem which actually writes to the display. In this way, a level of abstraction is provided between the client and the system so that customization can be facilitated without requiring the client to have a detailed knowledge of the interface environment, which may be constantly changing.

According to exemplary embodiments of the present invention, data structures for the layout resources are designed to allow the layout engine to operate efficiently. For example, a layout resource is specified so that the parts are redrawn correctly when the parent rectangle changes size. This type of functionality is provided by layout boundaries. The layout boundaries for each part identified in the layout resource can be placed on the display relative to a parent shape, e.g., a rectangle. When the layout engine is called to either draw or create a region, it first places the boundaries stored in the data structures associated with the layout resource. The location of the boundary can be calculated and stored so that when a part is drawn relative to the boundary, the part can be drawn on the screen in the right place.

Once the boundaries are placed, then the parts list is traversed by the layout engine. The layout engine checks to see if the part is a member of the set which is currently being drawn. If that part is not a member of the set, which can occur since layout resources can be used to draw different types of window objects, for example, then the listed part is not drawn. If the part is a member of the set being drawn, the attributes provided within the layout resource data structure for that part are checked to see if the part is valid. If the part is a member of the correct set and it is valid, then the part is drawn at the position specified by its previously placed boundaries.

A significant advantage of the data structures described above is that they are organized as a list of boundary parameters and part parameters which need only be traversed once by the layout engine to create the associated object. This provides benefits in terms of execution speed when rendering objects on the user interface.

## BRIEF DESCRIPTION OF THE DRAWINGS

The foregoing, and other, objects, features and advantages of the present invention will be more readily understood by those skilled in the art upon reading the following detailed description in conjunction with the drawings in which:

FIG. **1** shows a conventional desktop screen;

FIG. **2A** shows a conventional document window;

FIG. **2B** illustrates a document window according to an exemplary embodiment of the present invention;

FIG. **2C** illustrates a conventional user interface;

FIG. **2D** illustrates the user interface of FIG. **2C** operating under a theme according to an exemplary embodiment of the present invention;

FIG. **2E** illustrates the user interface of FIG. **2C** operating under a second theme according to another exemplary embodiment of the present invention;

FIG. **3** illustrates a functional overview of a system for customizing a user interface according to an exemplary embodiment of the present invention;

FIG. **4** illustrates an exemplary architecture showing theme and application interaction according to an exemplary embodiment of the present invention;

FIG. **5A** illustrates a block diagram representation of various layers used to provide a user interface according to an exemplary embodiment of the present invention;

FIG. **5B** illustrates the use of boundaries according to an exemplary embodiment of the present invention;

FIG. **5C** is another illustration which describes the use of boundaries according to exemplary embodiments of the present invention;

FIG. **5D** depicts variations in rendering a title bar for different window size conditions;

FIG. **5E** is a flowchart used to describe operation of an exemplary layout engine according to an exemplary embodiment of the present invention;

FIG. **6** is a state diagram used to illustrate transitions of an interface object part according to an exemplary embodiment of the present invention;

FIG. **7** is an exemplary matrix used to describe behavior transitions according to exemplary embodiments in the present invention;

FIG. **8** is a block diagram illustrating inheritance according to an exemplary embodiment of the present invention;

FIG. **9** is a block diagram which illustrates pattern abstraction according to an exemplary embodiment of the present invention;

FIG. **10** is a block diagram which also illustrates pattern abstraction, but according to another exemplary embodiment of the present invention;

FIG. **11** illustrates an exemplary appearance control panel according to an exemplary embodiment of the present invention;

FIG. **12** illustrates an interaction between an appearance management layer, an application, and a theme according to an exemplary embodiment of the present invention; and

FIGS. **13–15** are flowcharts which illustrate exemplary methods used to switch themes according to exemplary embodiments of the present invention.

## DETAILED DESCRIPTION

The present invention is described herein by way of exemplary, illustrative embodiments, some of which use the Macintosh® computer system as a reference for explaining the present invention. However, those skilled in the art will readily appreciate that systems and methods according to the present invention can be applied to any type of display

system having a user interface. Further, while window objects are used to illustrate how exemplary embodiments of the present invention affect the appearance and behavior of desktop objects in general, those skilled in the art will recognize that the present invention can be used to control the appearance and behavior of any desktop object including, for example, icons, menus, lists, control elements, cursors, menu bars, etc.

Windows can be characterized in a variety of ways. For example, a window can be characterized by the shape, size and color of the window as well as by the location, size, shape and color of its component parts, e.g., those parts identified in FIG. **2A**. These attributes of a window and window parts are categorized herein as a window's appearance attributes. The window and its parts also have associated therewith one or more functions which are invoked when a user provides an associated input, e.g., clicking on a close button or box causes the window to close. These are termed functional attributes.

A third category of attributes also exists for some windows and window parts. These windows and window parts exhibit a behavior when acted on by a user which is distinct from the underlying function of these objects, i.e., when a user clicks on a close button using a mouse, the button becomes shaded in such a way that it appears depressed prior to the window actually closing. These are termed behavior attributes.

Of these three attribute categories, namely appearance, behavior and function, exemplary embodiments of the present invention provide users (the term "users" as applied throughout this document refers to, among others, end users of applications, application developers and other individuals who use or invoke operating systems) with the capability to alter the appearance and behavior of object and object parts, but preferably not the underlying function thereof. It will be understood by those skilled in the art that the principles described herein are equally applicable to systems and methods in which the functional attributes can also be varied by users. However, standardization of system functionality provides certain advantages so that exemplary embodiments of the present invention separate functional manipulation from manipulation of the other attributes.

Given all of the graphical and audio artistry available today for GUIs, one can easily imagine the wide variety of desktop "looks" which can be developed once the system's control over the appearance and behavior of interface objects is relaxed. Comparison of the conventional user interface screen shown in FIG. **2C** with user interface screens using different themes shown in FIGS. **2D** and **2E** is an excellent starting point toward understanding the powerful capabilities for appearance and behavior change in user interfaces according to the present invention. Note, for example, the difference in appearance between the "Views" title bar in FIG. **2C** as opposed to those of FIGS. **2D** and **2E**.

An overview which summarizes how these types of customized user interfaces can be provided in a consistent and switchable manner begins with a discussion of FIG. **4**. As shown, the application **38** interacts with the appearance management layer **40** through three paths: directly, through utilities **42** (e.g., Toolbox Managers), and through drawing procedures **46** which provide the fundamental instructions (e.g., defprocs) for drawing objects on the interface. The phrase "drawing procedure" as it is used in this document refers to pieces of code which are responsible for drawing interface objects and which define the shape of those objects, e.g., window definitions.

Note that the application does not access the drawing procedures directly, but does so through a table of pointers **44** maintained by the appearance management layer and utilities. Switchable pointers **44** and drawing procedures **46** provide the basic building blocks which allow the geometry of each interface object as well as the behavior of each object's controls to be manipulated in a consistent and replaceable fashion. By switching the pointers **44** to the drawing procedures **46**, or by switching the data used by the procedures **46**, the appearance and behavior of the interface can be readily changed.

To provide the flexibility afforded by the present invention, applications do not need to have a priori knowledge of the patterns or colors used for each object and its controls. Therefore, a pattern table **48** is used to look up this information and serves to abstract the color and/or pattern of the object from its other attributes. According to certain exemplary embodiments, drawing primitives which allow "paint-by-number" interface drawing are sent by the client to the appearance management layer. In other words, the application can simply command the appearance management layer **40** to draw an object using an index which identifies the pattern and/or color of that object, so that the visual geometry is abstracted from the colorspace and the application need not know which particular geometries and/or colors are currently being implemented. According to other exemplary embodiments, the pattern table **48** acts as a pattern/color database and returns the relevant pattern information to the client. The client then instructs the graphic subsystem **56** to render the appropriate pattern.

In order to provide the functionality to switch between themes, the theme switching **50** and run time support **52** control interaction of the appearance management layer and the theme. As used herein, the terms "theme" and "themes" refer to coordinated designs of interface objects and object parts that create a distinct visual appearance on the display. These routines provide mechanisms for loading and unloading themes and obtaining theme attributes. Various routines are also provided to support animation and sounds and handling desktop patterns and screen saver modules in the interface as shown generally by block **54**.

### Switchable Pointers and Drawing Procedures

Many of the objects which are drawn in the user interface are created by small, modular pieces of code in the system which are dedicated to a specific purpose, e.g., drawing window frames. These pieces of code, called drawing procedures or definition procedures (defprocs) herein, are designed for switching at run time to enable dynamic system appearance and behavior. While the procedure and mechanism for switching between themes is described in more detail below, this section focuses on exemplary ways in which these procedures are designed to provide a switchable routine environment.

The appearance management layer **40** is responsible for orchestrating various changes which allow switching of the user interface's appearance and behavior. Two exemplary ways in which the drawing procedures can be switched will now be described here.

According to certain exemplary embodiments, all of the utilities which support switchable drawing procedures will be called to "disconnect" all of the drawing procedures for each of the interface objects supported by that particular utility. In essence, this amounts to sending a dispose message to the drawing procedure for utility object elements currently in existence. The utility then is called to swap

pointers **44** to the drawing procedures. For example, if window drawing procedure A is being replaced by window drawing procedure B, the window drawing utility will be asked to replace all of its references to procedure A with references to procedure B. This process will occur for each drawing procedure that is switched out. Lastly, every drawing procedure for every utility interface element should be sent an initialize message and the display will be completely redrawn.

According to other exemplary embodiments of the present invention, these drawing procedures can be data driven so as to allow each procedure to be able to support a wide variety of appearances and behaviors without modifying the code of the procedure itself. In this way themes can be switched without requiring that the drawing procedure code be switched. Each theme provides its own data structures which are supplied to the parametric drawing procedure. These exemplary embodiments will now be described in more detail.

According to certain exemplary embodiments of the present invention, system-provided drawing procedures map directly from existing procedures to provide compatibility with existing systems. For example, each individual drawing procedure will correspond to a conventional procedure (e.g., WDEF0, WDEF1, CDEF0, CDEF1). This mapping can be accomplished, for example, by the exemplary mapping procedure illustrated below in pseudocode form. This exemplary procedure can handle loading both conventional drawing procedures as well as the new drawing procedures.

```
OSErr MapDefprocReference(ResType defprocType, SInt16 defprocID,
                          Handle *defprocHandle, SOMObject **ido)
{
    OSErr result;
    //
    // First load the defprocType, defprocID resource
    //
    *defprocHandle = GetResource(defprocType,defprocID);
    //
    // If the resource came from the system, this identifies it as a stub
      and so get the corresponding ido pointer.
    //
    if (the Handle is a System Resource)
    {
    result = GetSystemIDO(defprocType,defprocID,ido);
    }
    else
    //
    // If the resource didn't come from the system, assume it's a
    // custom resource deproc and return a NULL ido pointer.
    //
    {
    result = noErr;
    *ido = NULL;
    }
}
```

The first step, as seen above, is to determine the resource ID of the procedure being called. This will load either an old style procedure located in a resource chain or a stub resource from the system file. Stub resources are modules which, when invoked, decode a conventional procedure's message and call the corresponding new drawing procedure based on the decoded message. Thus, when a utility creates a new interface object using a drawing procedure it will also load an appropriate stub resource and store its value in a procedure handle field of the object's data structure. Since the utilities can switch the drawing procedure that they call, the ability to dynamically change the set of drawing procedures which create the interface objects is now available.

## Data Driven Layout Engine

According to other exemplary embodiments of the present invention, the drawing procedures can be parametric in nature so that they need not be switched every time that a theme is changed. Instead, the data supplied to these procedures is changed with the theme. A discussion of data-driven layout engines according to the present invention begins with some background on window and menu definition objects.

Window and menu definition objects are modular pieces of code which define how window and menu interface elements look (i.e., are drawn on the screen) and react (e.g., respond to user mouse clicks). The following examples use window objects, however those skilled in the art will appreciate that other objects, e.g., menu objects, can be created using similar techniques. The system uses the window definition objects to provide graphical user interface interaction with a user. These definition objects can flexibly define windows in a variety of ways. For example, rectangular windows commonly seen in GUIs could relatively easily be replaced by round windows simply by writing a new window definition object.

There are several tasks that a window definition object typically performs. For example, the window definition object provides the capability to draw a window at any size anywhere on the screen. This code can also draw several different kinds of windows, such as windows with and without close boxes and zoom boxes, as well as dialog boxes and drawers. As mentioned above, replacing rectangular windows with round windows would conventionally require writing software (i.e., a new window definition object). One of the objects of exemplary embodiments of the present invention is to allow windows and menus to be changed without writing new code. Thus, there is described herein a single window definition object which evaluates data to determine how to perform tasks which were conventionally performed by various different window definition objects. The data can then be changed to make windows look and act differently, rather than changing the code.

To obviate the job of creating new code for each new window, the data-driven (switchable) behavior of window objects is incorporated into a general data-driven layout engine according to the present invention. The layout engine receives data resources and performs a system-generated drawing primitive. These drawing primitives, which are described in more detail below, can for example, draw an object or parts of an object at any size and place on the display. The layout engine can also create a region which corresponds to any set of the object's parts. FIG. 5A is a block diagram depicting command and data connections between various layers according to exemplary embodiments of the present invention. Therein, a user interface **500** is illustrated as interacting, e.g., by way of application program interface (API) calls, with both a layout engine **510** and a window definition object **520**.

When the window definition object **520** receives a command from the user interface **500** to draw a window, the window definition object **520** uses the drawing primitives, and appropriate data resources from resource manager **530**, to instruct the layout engine **510** to draw a window. After the layout engine **510** is finished executing its assigned drawing primitive(s), the window definition object **520** may perform any remaining tasks with respect to drawing the window. For example, the layout engine **510** will not draw the window title if it is not provided with the text of this title. According to this exemplary embodiment, the window title is not

passed from the window definition object **520** to the layout engine **510** to simplify communication. Nonetheless, the layout engine **510** has control over where the window title is drawn even if it doesn't draw the title itself, because the window definition object **520** will query the layout engine **510** for a region that corresponds to the window title text and then draw the title in that region. Other exemplary interactions between the layout engine **510** and window definition object **520** include the following.

- To draw the outline of the window, e.g., when a window is being grown, the window definition object **520** asks the layout engine **510** for the region of the window's structure. The window definition object **520** calls a routine, e.g., from the drawing procedures **46**, to draw the outline of the identified region.
- The window definition **520** object can perform a test to determine if a user has activated, e.g., clicked on, a particular part of a currently displayed window by asking the layout engine **510** for all the regions of all the window sets. Sets correspond to areas of a window such as close boxes, drag regions and zoom boxes. The window definition object **520** can then call a routine which checks to see if a pointer has been activated in each region.
- Drawing specific window elements, such as a size box or a highlighted window control, is performed by sending a drawing primitive from the window definition object **520** to the layout engine **510** identifying the part with its respective data resource.

Having described various exemplary functions which can be performed by the layout engine **510** upon command by the window definition object **520**, two exemplary primitives which can be used to convey these commands will now be described. Exemplary syntax for a DrawLayout primitive and a RegionLayout primitive are set forth in Table 1 below.

TABLE 1

```
OSErr DrawLayout(UInt32 LayoutRef, Rect portRect, UInt16 Set,
UInt16 Metavalues, UInt32 Attributes)
OSErr RegionLayout(UInt32 LayoutRef, Rect portRect, UInt16 Set,
UInt16 Metavalues, UInt32 Attributes, RgnHandle theRgn).
```

Each of these commands have certain parameters which are passed to the layout engine **510** from the window definition object **520**. For example, the LayoutRef parameter is a pointer to the layout data. Several examples and more details regarding the layout data and its associated data structures are provided and discussed below. The window definition object **520** passes the location of the layout data from, e.g., the appearance management layer **40** of FIG. **4**, to the layout engine **510** using the LayoutRef pointer.

The portRect parameter identifies a parent rectangle around which the layout engine **510** draws the parts on the display. The graphics subsystem **56** can specify the portRect parameter for every operation. The window definition object **520** receives the portRect parameter from the system and passes this parameter on to the layout engine **510**.

The Set parameter identifies the set of parts which the layout engine **510** is to draw. There can be a predetermined number of possible sets defined for a window (e.g., 32 sets) which are predefined by the window definition object **510** for different regions. The menu definition object can define a different list of sets which would be used to draw menus. As an example, the window definition object can use the following sets associated with different regions of a window:

- kSetTitleBar—draws the title bar at the top of the window.
- kSetTextRgn—determines the area where the title text should be drawn.
- kSetCloseBox—draws the window close box.
- kSetZoomBox—draws the window zoom box.
- kSetIconBox—determines the area where the window title icon should be drawn.
- kSetCustomBox—determines the area where the custom icon should be drawn.
- kSetDrag—determines the area where a user can click on the window to drag it.
- kSetGrow—draws the window grow box.
- kSetStructure—draws the structure of the window (for instance, the left and right sides and bottom).
- kSetContent—determines the area where the content of the window is drawn.

The Metavalues parameter is an arbitrary length array of extra values which the window definition object **520** can define and which can then be used by the layout engine **510** to size the parts to be drawn on the display. For instance, the window definition object **520** can pass values such as the title text height and width in this array.

The Attributes parameter is a bit field of flags which are provided by the window definition object and are used to limit which parts are considered valid for a particular primitive. A list of parts, described below, is provided as part of the layout resources. The window definition object defines which flags are used in this field. For instance, the Attributes parameter can define the following bits: kAttributeHasGrowBox, kAttributeRightToLeftOrientation, kAttributeHasFullZoom, kAttributeHasCloseBox, kAttributeHasCustomGadgetIcon, kAttributeHasTitleText. An illustrative example will serve to highlight the functions and advantages of the Attributes parameter.

Suppose that the window definition object **520** called the DrawLayout primitive of Table I with Set equal to kSetTitleBar and Attributes equal to kAttributeHasCloseBox, kAttributeHasTitleText, kAttributeHasFullZoom, e.g., by setting bits in the mask associated with these attribute parameters equal to "1". Then, the layout engine **510** would draw a title bar with a close box, title text and a zoom box. If the window definition object **520** subsequently called DrawLayout again with the same arguments (including a LayoutRef parameter which pointed to the same layout data) except that for this iteration Attributes equals kAttributeHasCloseBox and kAttributesHasCustomGadgetIcon, the layout engine **510** would draw a title bar with a close box and a custom gadget icon but with no zoom box or space for title text. The Attributes parameter provides added flexibility to the DrawLayout primitive such that the window definition object **520** can use a single layout resource to describe a variety of different windows which have different attributes.

As can be seen from the syntax in Table 1, the RegionLayout routine uses the same parameters as the DrawLayout routine plus an extra parameter, theRgn, which identifies the region which is returned. The purpose of the DrawLayout primitive is to draw something on the screen, while the purpose of the RegionLayout primitive is to create a region (shape) and return that region to the window definition object **520** so that the window definition object can operate on the region. For instance, the window definition object could then draw the outline of the region on the screen itself by calling a framing function from the graphics subsystem. Alternatively, the window definition object **520** could draw the window title at the spot on the screen specified by the region returned by RegionLayout.

To draw a window, the window definition object **520** loads the layout resource for the particular type of window

which is to be drawn. Alternatively, the layout resources can be pre-loaded and simply pointed to by the window definition object **520** using the LayoutRef parameter. Then the window definition object **520** sets the Metavalues array to the correct values for the current window title height and width. Finally, the window definition object **520** calls the layout engine **510**, for example by instructing:

DrawLayout(LayoutRef, portRect, kSetStructure, Metavalues, kAttributeHasCloseBox);

The foregoing command provides an example of a specific DrawLayout command. The parameter kSetStructure specifies to the layout engine **510** that the window definition object **520** wants the window's structure region drawn, e.g., the entire outline of the window including the title bar, the close box, zoom box and title text region. By specifying the parameter kAttributeHasCloseBox the window is drawn by the layout engine **510** with a close box (and without a zoom box, etc., since other Attributes are not specified in the drawing primitive).

Having provided an overview of how data-driven layout engines operate according to the present invention, a more detailed description of layout resources will now be provided. A layout resource is a data structure which includes a list of boundaries and parts. The boundaries contain the information regarding the location on the display in which the parts should be drawn relative to a reference shape, e.g., parent rectangle. The parts contain the information about which pattern is drawn in each part rectangle. There can be any number of boundaries and any number of parts in each layout resource. The following is an exemplary data structure for a layout resource.

```
struct ThemeLayout{
UInt16          version;
UInt32          reserved;
UInt16          boundaryCount;
ThemeBoundary   boundary[1];
UInt16          partCount;
ThemePart       part[1];
};
```

The version parameter provides version control to allow a theme designer to indicate for which version of the layout engine this particular layout resource was created. The reserved parameter is a place holder for future functionality. The boundaryCount parameter indicates how many ThemeBoundary data structures follow in the list.

Exemplary ThemeBoundary data structure implementations for providing boundary information in the layout resource data made available to the layout engine by the drawing primitive will now be discussed. For example, a single boundary having a fixed length structure can take the format described below.

```
struct ThemeBoundary {
UInt16          offsetType;
SInt16          value;
SInt16          startBoundary;
UInt16          limitType;
SInt16          limit;
SInt16          LimitBoundary;
};
```

As mentioned previously, each boundary is placed on the screen relative to any of the previously placed boundaries. In the above example, the startBoundary parameter specifies the relative position and orientation of the new boundary. For example, if startBoundary is a horizontal boundary, then the new boundary will be a horizonal boundary also.

The offsetType parameter can have one of the following values: kOffsetConstant, kOffsetMetaValue, and kOffsetCenter. The value parameter has different meanings depending on what value the parameter offsetType takes. Specifically, if the offsetType parameter is kOffsetConstant, then the value parameter is a positive or negative number which is added to the startBoundary location to obtain the current boundary location. For example, to create a new boundary which is six pixels to the right of the left side of the parent rectangle, one could specify the left side of the parent rectangle as the startboundary, set offsetType to be kOffsetConstant, and give the value parameter a value of six. In the alternative, to create a new boundary which is six pixels to the left of the right side of the parent rectangle, one could specify the right side of the parent rectangle as the startBoundary, set offsetType to be kOffsetConstant, and give the value parameter a value of negative six.

If the offsetType parameter is specified as kOffsetMetaValue, then the value parameter functions as an index into the MetaValues array which was passed to the layout engine **510**, by the window definition object **520**. The absolute value of the value parameter is used to index the array. However, if the value parameter is negative, then the value stored in the MetaValues array at the index ABS (value) is negated before it is used. The value retrieved from the array (or its negative) is then added to the startboundary parameter to obtain the new boundary location as in the previously discussed case where offsetType was set to kOffsetConstant. If the offsetType is kOffsetCenter, then the value parameter is ignored, and the new boundary is placed halfway between the startBoundary and the limitBoundary.

Once the boundary has been placed using the above rules, the location can be further modified by limit restrictions as defined by the limitType parameter value. If the offsetType parameter is kOffsetCenter or the limitType parameter is kLimitNone, then the limit parameters have no effect on the boundary position, so the next boundary to be placed is processed. If the offsetType parameter is not kOffsetCenter and the limitType parameter is not kLimitNone, then the new boundary location can be modified based on the value of limitType, which can have, for example, any of the following values: kLimitNone, kLimitPinToLimit, LimitPinToStart, kLimitPinToLimitPeriod, and kLimitPinToStartPeriod.

If the limitType parameter is kLimitPinToLimit or kLimitPinToStart then this exemplary technique will consider the limitboundary parameter, which is another of the previously positioned boundaries. An exemplary diagram which portrays relationships between exemplary boundaries is provided as FIG. **5B**. The limitboundary limits movement of the new boundary **572**. An offset constant, for example, limit in FIG. **5D** (which can be positive or negative), defines an offsetlimitboundary **574**. The value of limit can be zero, in which case, offsetlimitboundary **574** and limitBoundary **570** would be the same. If the new boundary is to be placed in a position such that it has moved from the startBoundary to its new position and passed the offsetlimitBoundary, then it will be constrained based on the value of limitType. If the limitType parameter is set to kLimitPinToLimit then the new boundary **572** will be moved to be in the same place as the limitBoundary **570**. If limitType is kLimitPinToStart then the new boundary will be moved to be in the same place as the startBoundary **576**. Of course, if the new boundary would not pass the offsetlimitboundary when it is placed, as

is the case for new boundary **572** in FIG. 5B, its original placement will remain unchanged.

Two other values which the limitType parameter can take are kLimitPinToLimitPeriod and kLimitPinToStartPeriod. These are similar to the other two values of limitType except that they provide a series of offsetlimitboundaries spaced periodically. The new boundary is then moved to the closest offsetlimiboundary which is either in the direction of the startBoundary or the limitboundary depending on the value of limitType as kLimitPinToStartPeriod or kLimitPinToLimitPeriod, respectively. An example of this type of boundary placement is illustrated in FIG. 5C, wherein the offsetlimitboundaries are placed at equal spacings of limit from the limitBoundary. For instance, the first offsetlimitboundary **580** is limit pixels to the left of limitBoundary **582**. The second is limit pixels to the left of the first offsetlimitboundary **580**. The third offsetlimitBoundary **586** is limit pixels to the left of the second offsetlimitboundary **584**, etc. There is also another string of these offsetlimitboundaries to the right of limitboundary **582** as represented by offsetlimitboundary **588**. For the value shown in FIG. 5C, the new boundary **590** will be placed at offsetlimitboundary **584** if limitType is equal to kLimitPinToStartPeriod or at offsetlimitboundary **580** if limitType is equal to kLimitPinToLimit. An example of how the boundaries determine the visual appearance of objects and object parts is provided below with respect to Figure 5D.

Having described how boundaries can be structured in a layout resource, part structures will now be described. Each layout resource will include any number of parts as specified by the partCount parameter in the layout data. According to this exemplary embodiment, a part can be a fixed length structure having the following general form.

```
struct ThemePart {
    UInt32      inclusionAttributes;
    UInt32      exclusionAttributes;
    UInt32      regionSet;
    SInt16      top;
    SInt16      left;
    SInt16      bottom;
    SInt16      right;
    OSType      fillType;
    SInt16      fillID;
    UInt16      fillAnchor;
};
```

Each of the exemplary part parameters shown above will now be described below.

The inclusionAttributes and exclusionAttributes masks allow the part to specify that it should be drawn only when a certain attribute bit (e.g., kAttributeHasCloseBox) is present, or that it should be drawn only when a certain attribute bit is not present. There may be many bits set in each mask. As an example, if a certain bit (e.g., kAttributeHasCloseBox) is not set in either the inclusionattributes mask or the exclusionattributes mask of a particular part in a set to be drawn, then that part is present regardless of whether the layout engine is drawing a window with a close box or not. If the bit is set only in the inclusionattributes mask, then that part will only be drawn if the window has a close box. If the bit is set only in the exclusionAttributes mask, then that part will only be drawn if the window does not have a close box. It is illegal to have a particular bit set in both the inclusionAttributes and exclusionattributes masks.

The regionSet parameter specifies to which sets a part belongs. A single part may be a member of many sets. For example, if a single layout resource is to define how a window should look both with and without a close box, the layout resource would then contain at least two parts. One part would draw a close box pattern and have the kAttributeHasCloseBox bit in inclusionAttributes marked. Another part would be a filler pattern which would be drawn in place of the close box and have the kAttributeHasCloseBox bit in exclusionSet marked. Both of these parts could have the bit marked in their regionset masks indicating that they are a member of the kSetCloseBox.

The top, left, right and bottom parameters can be index numbers into the boundary array which is described above or they can be special numbers referencig the parent rectangle. The four boundaries form a rectangle which defines a location at which the part should be drawn on the display. The boundaries are placed on the display relative to the parent reference rectangle specified by the function call in the portRect parameter.

The fillType and filmID parameters specify how the part should be painted. For example, the fill could be a color pattern. The fillAnchor parameter specifies how the fill pattern should be oriented with respect to the part. This parameter can take, for example, one of the following values: kAnchorNone, kAnchorTopLeft, kAnchorTopRight, kAnchorBottomLeft, and kAnchorBottomRight.

For instance, if the part rectangle was too small for the fill color pattern, a value of kAnchorTopLeft would indicate that the color pattern should be clipped on the bottom and right. Also, if the part rectangle is sufficiently large that the fill icon is to be tiled (i.e., repeated) to fill the entire part rectangle, a value of kAnchorTopRight would indicate that the boundaries of the tiled color patterns should line up with the top and right boundaries of the fill rectangle.

This exemplary data structure for the layout resource allows the layout engine **510** to operate efficiently. For example, according to this exemplary embodiment of the present invention, a layout resource is specified so that the parts are redrawn correctly when the parent rectangle changes size. For instance, if a window is reduced in size, the graphics designer may want the title bar racing stripes in the window title bar to get shorter while maintaining the full title text width. This is shown in Figure 5D by way of title bars **550** and **552**. Once the window gets smaller than a certain size, however, the title text should be clipped while the length of the title bar racing stripes remains constant as illustrated by title bars **554** and **556**.

This type of functionality is provided by the layout boundaries described above. Consider the boundaries **560**, **562**, **564** and **566** illustrated in Figure 5D. These boundaries are specified in the layout resource for this title bar. Boundaries **560** and **562** limit how short the racing stripes can become. Boundaries **564** and **566** are centered about **568** to define the title bar text area. When the title bar shrinks to the size of title bar **552**, the limit boundaries **560** and **562** have not been exceeded by boundaries **564** and **566**, so the title bar text area is unaffected. However, when the title bar shrinks to the size **554**, the limit boundaries **562** and **564** become the boundaries **564** and **566** of the text area and force the text to be clipped.

The layout boundaries for each part identified in the layout resource can be placed on the display relative to the parent rectangle according to the exemplary embodiment illustrated by the flowchart of Figure SE. When the layout engine is called to either draw or create a region, it first places the boundaries as shown by block **560**. The location of the boundary can be calculated and stored so that when a part is drawn relative to the boundary, the part can be drawn

on the screen in the right place but the boundaries themselves are not drawn on the screen.

Of course, the boundaries should be determined quickly and can, for example, be placed linearly one at a time starting at the beginning of the boundary list. All of the boundaries can be placed on the screen every time the layout engine **510** is called by the window definition object **520**. Additionally, the boundary positions can be cached so that in certain circumstances, e.g., when the layout engine **510** is called with repeated parameter values such that the boundaries are in the same place the second time, the old boundary positions can be recalled from memory, which is faster than placing all the boundaries on the screen again. New boundaries are placed on the screen relative to the existing boundaries. When the first boundary is placed, the parent rectangle has already been placed on the screen thus, the four sides of the parent rectangle can be considered to correspond to boundaries having, for example, indices −1, −2, −3 and −4.

Once the boundaries are placed, then the parts list is traversed beginning with a first indexed part at block **562**. Then, at decision block **564**, the layout engine checks to see if the part is a member of the set which is currently being drawn by evaluating the aforedescribed regionSet parameter of that part. If not a member of the set, then the part is not drawn and the flow returns to block **562** to evaluate the next part in the layout resource. If the part is a member of the set being drawn, following the "Yes" branch in FIG. **5**E, then the attributes are checked at block **566** to see if the part is valid. If either the inclusionattributes or exclusionAttributes mask specifies that the part is invalid, it is ignored. If the part is a member of the correct set and it is valid, then the part is drawn where its four previously placed boundaries specify at block **568**. A significant feature of the data structures described above is that they are organized as a list of boundary parameters and part parameters which can be traversed once by the layout engine **510** to create the associated object. This provides benefits in terms of execution speed when rendering objects on the user interface.

To further describe how exemplary embodiments of the data driven layout engine operate, a simplified numerical example of layout data which can be used by the layout engine **510** according to the present invention is provided below where the values, in order, represent:

| Description | Value |
|---|---|
| version number | 0 |
| reserved | 0 |
| boundary count | 2 |
| kOffsetCenter - boundary 1 | 3 |
| value (ignored in this case) - boundary 1 | 0 |
| 1st boundary - boundary 1 | −1 |
| limitType (ignored in this case) - boundary 1 | 0 |
| limit (ignored in this case) - boundary 1 | 0 |
| 2nd boundary - boundary 1 | −3 |
| kOffsetConstant - boundary 2 | 1 |
| value - boundary 2 | 10 |
| startBoundary - boundary 2 | 1 |
| kLimitPinToLimit - boundary 2 | 2 |
| limit - boundary 2 | 5 |
| limitBoundary - boundary 2 | −3 |
| part count | 1 |
| kAttributeHasCloseBox inclusionAttributes | 1 |
| kAttributeHasZoomBox exclusionAttributes | 8 |
| contentRgn & structureRgn regionset | 3 |
| top | 1 |
| left | −2 |
| bottom | 2 |

-continued

| Description | Value |
|---|---|
| right | −4 |
| filltype | 'plut' |
| fillID | 5 |
| kAnchorTopLeft | 2 |

This simplified resource can be used to draw a rectangle, one edge of which is centered in the parent rectangle. The rectangle is 10 pixels wide and its other edge is limited to five pixels.

According to further exemplary embodiments, layout engine speed is increased since it will frequently be in use in the user interface. One of the slowest operations in many systems, e.g., the Macintosh® computer system, is allocating memory. Unfortunately, memory is frequently allocated when regions are being created, for example, when windows are drawn on the user interface. Thus, two additional features can be provided to the layout engine **510** to give the window definition object **520** the capability to perform rendering operations without having to create regions.

For example, it was stated above that the window definition object **520** can query the layout engine **510** for the region which corresponds to the window title text so that after the layout engine has drawn the window, the window definition object can draw the title text in the correct location. However, the window definition object **520** only needs to know the rectangle into which it should draw the title text, not the region since a region can be any shape, not just rectangular. Thus, an additional command call can be provided which is similar to RegionLayout described above except that it returns a rectangle which encloses the desired set. An exemplary format for this call is described below.

OSErr RectLayout(UInt32 theLayoutRef, Rect portRect, UInt16 Set, UInt16 Metavalues, UInt32 Attributes, Rect *bounds);

The second category of data structures used in the data driven structural procedure relate to interface objects' behaviors. Each behavior is associated with transitions between different states or values of controls in the interface and can be expressed by changes in visual or audio output that correspond to these transitions.

Data driven drawing procedures can use a common mechanism that implements state tables. These state tables contain bitmaps or glyphs for each state of the control represented thereby as well as information about transitions from one state to another. Each transition may contain one or more of, for example, an animation sequence, a sound or a routine to implement a custom transition, e.g., an algorithmic display or any other type of transitional effect. By defining state diagrams for each object and object part of the user interface, a template can be created that allows a theme designer to place customized glyphs for each state of the control and also to customize the transitions between states of the control as desired. An exemplary state diagram is shown as FIG. **6** which provides an example of the possible states and most common state transitions for a checkbox control of a window object.

As seen in FIG. **6**, this exemplary checkbox has nine possible states which can be displayed. These states include three highlighted states for each of the control's three values. In normal use, when a user clicks on an unchecked checkbox (state Q1), this action moves the control to its pressed state (state Q4). After the mouse is released, the control returns back to its original state (state Q1) and the application is

notified of the button which has been pressed. The application then switches the value of the control to its new value, which might be checked (state Q2).

In data driven themes according to the present invention, a resource exists for each of the customizable controls to allow the theme designer to plug in new glyphs or bitmaps for each of the states of the control. Moreover, to provide more flexibility to customize transitions between states and a control state table, a matrix for these transitions can be provided. Note for example the exemplary matrix illustrated in FIG. **7**. For each block in the matrix, a theme designer can provide a visual and/or audio output such as an animation, sound, a custom transition procedure which can perform some type of algorthmic transition, e.g., a kaleidoscopic display or any combination thereof. Of course, not every box in the transition matrix need be filled in by the theme designer and where no transition behavior is specified, or if the theme does not specify a special transition behavior, the control moves directly to the glyph or bitmap that is specified for the new state without any transitional effect.

Although the foregoing two exemplary embodiments describe switching either the code or the data of the drawing procedures, those skilled in the art will appreciate that both schemes can be implemented in the same interface. For example, it may be advantageous to generate certain themes, e.g., themes using relatively simple patterns, by way of hard-coded drawing procedures to provide a speedy redrawing of the interface. Similarly, where a theme is, for example, relatively more complicated graphically, it may be advantageous to generate such themes using the aforedescribed data-driven drawing procedures. Accordingly, since many different types of themes are available for user selection, it is anticipated that both of the above-described exemplary embodiments can be deployed in the same interface and the switchable pointers will then either point to the appropriate hard-coded procedure or to the parametric drawing procedure.

Custom drawing procedures can inherit from the system provided appearance using a form known as delegation or forwarding. Delegation involves passing control on to another object when inherited behavior is desired. To determine the particular object to which the drawing procedure should delegate in a dynamically changing interface, either the client can call in to the system or the system can track current implementations. According to exemplary embodiments, this burden can be placed on the system by providing an additional layer of redirection. As seen in FIG. **8**, the utility **61** calls the custom drawing procedure **63**. The drawing procedure **63** inherits from the switcher **65** which delegates to the appropriate implementation **67** or **69**. An example of this type of inheritance will now be described using menu drawing procedures.

A theme can provide a menu drawing procedure which controls drawing standard menus for that theme for example, a theme may only change the appearance or behavior of a single item in the menu while letting the remaining menu items appear and behave as they do when the system default theme is in control. By creating a custom menu drawing procedure that inherits from the system menu drawing procedure, i.e., from the switcher object, the application can intercept the command to draw a menu item from the utility issuing the command. If the menu item to be drawn is an item whose appearance and/or behavior has been customized by the theme, then the theme's menu drawing procedure can be used to draw that item. Otherwise, the inherited code pointed to by the switcher object can be called to draw the item. In this particular example, where the theme only customizes one menu item, the theme's custom menu drawing procedure only overrides the system to draw that item, with any other items being drawn using the inherited code.

Pattern Look-up Tables and Drawing Support

The following discussion relates to the subject matter described in the above-referenced U.S. Patent Application entitled "Pattern and Color Abstraction in a Graphical User Interface." The following is a more detailed description of the pattern look-up table mechanism **48**. As described above, since one of the objects of the present invention is to provide interfaces which facilitate user control over the appearance of the desktop, the themes used by the appearance management layer **40** should be able to operate on a variety of color data to draw the interface, e.g., a color pattern, a pattern defined on a pixel-by-pixel basis, bitmapped image or the like, etc. The pattern tables provide a system and method for specifying this color data, so that the theme color set can be edited independently of the theme using resource-editing utilities. The pattern tables provide this support by abstracting the notion of pen pattern and color, allowing an application or theme to draw interface pieces without being locked to a particular color set.

This functionality is provided according to exemplary embodiments of the present invention by a mechanism including a pattern look-up table. An index in a pattern look-up table references data for a color, a pattern defined on a pixel-by-pixel basis, bitmapped image or the other data, so that the client need not know anything about the datatype contained in the pattern look-up table entry. The significance of this data independence is that a theme having solid-colored windows, for example, can be changed to instead draw the windows in a complex pattern, without changing the theme source code simply by editing the table entries. When reference is made below to the terms "pattern" or "patterns", it is intended to denote any type of graphic data that can be used in a pattern look-up table to draw in a graphics port. As such, this may be a solid color defined in terms of its red, green and blue (RGB) components, or a pattern defined on a pixel-by-pixel basis, e.g. a PixPat, or a new type of data.

Before discussing the various features of the pattern table routines in great detail, an overview of how color and pattern abstraction can be provided according to an exemplary embodiment will be described with reference to FIG. **9**. Therein a client **60** sends a command ThemeFillRect (kColorIndex) to the appearance management layer. This command is one of a set of drawing primitives implemented by the appearance management layer **40**. In this particular example, it is a command to draw a rectangle that is filled with the pattern specified as kColorIndex. The value of kColorIndex corresponds to a predetermined object or object part on the desktop. For example, index **3** might correspond to the window title color. However, note that the client **60** need have no knowledge of the particular color which is currently being implemented as the window title color, but only the absolute index which identifies that color.

The kColorIndex parameter has a corresponding entry in the part index table **62**. This entry maps into the theme pattern look-up table **64**. As described previously, the entries in the theme pattern look-up table **64** can include any type of color or pattern data in any format. For the purposes of this example suppose that the entry in the part index table corresponding to the value of kColorIndex maps into a pattern called 'xpat' referring to a black and white criss-

cross pattern. 'Xpat' has a corresponding entry in the pattern definition procedure table **66** where the procedure for drawing this black and white criss-cross pattern is located. This table includes a procedure pointer **68** which translates the commands defined by the 'xpat' record into commands which are recognized by the graphic subsystem **56** used by the system to draw the pattern onto the display. These commands are then sent to the graphic subsystem which displays the pattern at the appropriate point on the desktop interface.

Although the exemplary embodiment illustrated in FIG. **8** portrays the client as using drawing primitives to send commands through the appearance management layer to the graphic subsystem, other exemplary embodiments of the present invention operate in a somewhat different fashion. According to this exemplary embodiment, the appearance management layer **40** does not command the graphic subsystem **56**, but simply acts essentially as a pattern/color database. For example, in the exemplary block diagram of FIG. **10**, a get theme pattern command is sent to the appearance management layer **40**, instead of the drawing primitive in FIG. **8**. The appearance management layer returns a pattern structure which can be rendered by the graphic subsystem in the currently implemented theme for the particular interface object or object part requested in the get theme pattern command, to the client which then sends its own command to the graphic subsystem to draw the appropriate pattern and/or color on the desktop interface. This alternate exemplary embodiment also has the benefits described herein with respect to abstracting the pattern/color combination from the interface.

Thus, through the use of pattern tables, the color and/or pattern of desktop objects can be readily switched from one theme to another by changing the values in the part index table **62** and/or the pattern look-up table **64**. This switching of patterns is totally transparent to the application. As a result, new patterns can be added without any need to change the application itself. Having now described an overview of pattern and color abstraction according to the present invention, a more detailed description of exemplary routines for implementing the above will now be provided.

The appearance management layer, according to certain exemplary embodiments, recognizes a set of drawing primitives which can be, for example, derived from those used by the system's graphic subsystem (for example, QuickDraw). These primitives can have the same calling sequence as their counterparts in the graphic subsystem, but use indices into the theme pattern table to specify the color and/or pattern details of the requested drawing command. Exemplary drawing primitives are illustrated below along with descriptions in italics.

```
typedef unsigned char OSType [4];
typedef short SInt16;
typedef unsigned short UInt16;
typedef unsigned long UInt32;
typedef UInt16 ThemePartIndex;
```

pascal OSErr ThemeSetPen (ThemePartIndex);

Sets the pen pattern to the contents of the specified index of the theme pattern look-up table.

pascal OSErr ThemeFrameRect (ThemePartIndex, Rect *r);

pascal OSErr ThemeFillRect (ThemePartIndex, Rect *r);

Fills or frames the rectangle with the contents of the specified index.

pascal OSErr ThemeFrameRoundRect (ThemePartIndex, Rect *r, radius);

pascal OSErr ThemeFillRoundRect (ThemePartIndex, Rect *r, radius);

Fills or frames the round rectangle with the contents of the specified pattern index.

pascal OSErr ThemeFrameOval (ThemePartIndex, Rect *r);

pascal OSErr ThemeFillOval (ThemePartIndex, Rect *r);

Fills or frames the oval with the contents of the specified pattern index.

pascal OSErr ThemeFramePoly (ThemePartIndex, PolyHandle);

pascal OSErr ThemeFillPoly (ThemePartIndex, PolyHandle);

Fills or frames the polygon with the contents of the specified pattern index.

pascal OSErr ThemeFrameRgn (ThemePartIndex, RgnHandle);

pascal OSErr ThemeFillRgn (ThemePartIndex, RgnHandle);

Fills or frames the region with the contents of the specified pattern index.

The appearance management layer can also define a set of bevel, text and dialog grouping rectangle primitives which can be used by clients for drawing bevels and dialog group rectangles in a standard appearance. The implementations of these routines can be overridden by the theme to generate special appearances. For this reason, the client should not draw bevels independent of the appearance management layer for user interface object parts, but should instead use the provided primitives. Exemplary primitives are shown and described below.

pascal OSErr ThemeDrawBevel (Rect *pBevelRect, Boolean fbutton); pascal OSErr ThemeDrawInsetBevel (Rect *pBevelRect, Boolean fbutton);

Draws a bevel into or out of the background. If button is set, then the bevel corners are left out, resulting in a standard 'beveled button' visual.

pascal OSErr ThemeDrawDeepBevel (Rect *pBevelRect, Boolean fbutton);

pascal OSErr ThemeDrawDeepInsetBevel (Rect *pBevelRect, Boolean fbutton);

Draws a deep bevel into or out of the background.

pascal OSErr ThemeDrawInsetTextFrame (Rect *pTextFrame);

Draws the standard inset text frame which is used for edit text items in dialogs.

pascal OSErr ThemeDrawRidge (Rect *pRidgeRect);

pascal OSErr ThemeDrawInsetRidge (Rect *pRidgeRect);

Draws a ridge frame into or out of the surface.

pascal OSErr ThemeDrawEmbossedString (StringPtr, scriptcode);

pascal OSErr ThemeDrawInsetString (StringPtr, scriptcode);

Draws a string embossed out of, or inset into, the surface.

pascal OSErr ThemeDrawShadowedString (StringPtr, scriptcode);

Draws a string with a shadow.

pascal OSErr ThemeMeasureEmbossedString (StringPtr, scriptcode, Rect);

pascal OSErr ThemeMeasureInsetString (StringPtr, scriptcode, Rect *);

pascal OSErr ThememeasureShadowedString (StringPtr, scriptcode, Rect);

Measure the size of the string when embossed.

pascal OSErr ThemeDrawGroupingRect (Rect *pGroupRect, Str255grouptitle);

Draws a dialog item grouping rect with the specified title. An empty or nil title may be passed and no title will be drawn.

pascal OSErr ThemeDrawSeparatorLine (Int16 length, Boolean fvertical);

Draws a horizontal or vertical separator line.

Pattern look-up tables are provided as part of the package which handles drawing requests, either with the afore-described drawing primitives or alone, which tables will now be described in somewhat more detail.

A pattern data structure holds the data necessary to draw a pattern. It can have, for example, the following structure:

typedef UInt32 PatternData [2];

typedef PatternData *PatternDataPtr;

The pattern data structure can be, for example, an eight-byte structure used to store pattern and/or color information. If the data required for the pattern is more than eight bytes long, it can be stored in a handle and the handle placed in the pattern data structure. A pattern definition procedure, described below, is a component which is responsible for the loading and interpretation of a pattern data structure.

The pattern look-up table specifies the list of colors and patterns used by a theme. A pattern look-up table contains a list of records, e.g., Pattern Spec record 'xpat' in FIG. 9, each of which is typed and references a specialized procedure to load, unload and draw.

Data encapsulation within a pattern look-up table entry is accomplished through use of a pattern definition procedure, a code module responsible for loading, unloading and interpreting a pattern look-up table entry's data, e.g., the pattern definition procedure 'xpat' of block 66 in FIG. 9. New pattern types may be defined by a theme for specific needs, such as algorithmic color and pattern generation, simply by adding new pattern definition procedures. A pattern definition procedure can be defined, for example, as a code fragment module or a dynamically loaded library which exports a list of entrypoints as set forth below. The default behavior for unimplemented entrypoints is to return an error.

OSErr PatDefOpen (OSType *pPatternType);

Called when the pattern def is initially loaded, to allow the procedure to initialize state data. *pPatternType should be set to an OSType denoting the pattern type it will handle, for example 'xpat' or 'ppat'.

OSErr PatDefClose (0);

Called when the pattern def is no longer needed, to allow release of state data.

OSErr PatDefLoadData (PatternDataPtr, Int16 id, Int16 index);

Load the data associated with this pattern from a resource, and place the data in the PatternData record pointed to by PatternDataPtr.

OSErr PatDefSetData (PatternDataPtr, PatternDataPtr newdata);

Set the pattern data to a copy of that located in newdata.

OSErr PatDefFreeData (PatternDataPtr);

Free the data in the PasemData record pointed to by PattepDataptr.

Set the port's pen to draw with the pattern.

OSErr PatDerFrameRect (Pattey DataPtr, Rect *);

OSErr PatDefFrillRect (PatterDataPtr, Rect *);

Fill or frame the rectangle.

OSErr PatDefFrameRoundRect (PatterDataPtr, Rect *, UInt16 w, Unt16 h);

OSErr PatDefFillRoundRect (Pattea DataPtr, Rect *, UIn16 radius);

Fill or frame the rounded rectangle.

OSErr PatDefFrame0val (PatternDataPtr, Rect *prect);

OSErr PatDefFillOval (PatternDataPtr, Rect *prect);

Fill or frame the oval contained in the rect.

OSErr PatDefFramePoly (PatternDataPtr, PolyHandle hpoly);

OSErr PatDefFillPoly (PatternDataPtr, PolyHandle hpoly);

Fill or frame the polygon.

OSErr PatDefFrameRgn (Pattern DataPtr, RgnHandle rgn);

OSErr PatDefFillRgn (Pattern DataPtr, RgnHandle rgn);

Fill or frame the Range.

Pattern look-up tables may be created in m emory by applications to allow them the benefits of a patter look-up table within the application. An exemplary application program interface (API) for creating pattern look-up tables is described below.

typedef void *PatternTableRef;

typedef UInt16 PatternIndex;

pascal OSErr NewPatternSpecTable (PatternTableRef*);

pascal OSErr DisposePatternSpecTable (PatternTableRef);

Creates and Disposes a PatternSpecTable.

pascal OSErr AddPatternSpecToTable (PatternTableRef, OSType patternkind, PatternDataPtr pdata, PatternIndex *pindex);

Adds a new pattern spec to a PatternSpecTable. Patterns are always added to the end of the table. The index at which the pattern is placed is returned in pindex.

pascal OSErr GetPatternIndexType (PatternTableRef, PatternIndex, OSType *patternkind);

Returns the type of pattern located in the specified index of the table.

pascal OSErr SetPatternSpecData (PatternTableRef, PatternIndex, OSType pattenikind, PatternDataPtr pdata);

Set the pattern spec at the specified index to contain the specified data.

pascal OSErr PatternTableSetPen (PatternTableRef, PatternIndex);

Sets the pen pattern to the contents of the specified index of the theme pattern look-up table.

pascal OSErr PatternTableFrameRect (PatternTableRef, PatternIndex, Rect *r)

pascal OSErr PatternTableFillRect (PatternTableRef, PatternIndex, Rect *r);

Fills or frames the rectangle with the contents of the specified index.

pascal OSErr PattemTableFrameRoundRect (PatternTableRef, PatternIndex, Rect *r, radius);

pascal OSErr PatternTableFillRoundRect (PatternTableRef, PatternIndex, Rect *r, radius);

Fills or frames the round rectangle with the contents of the specified pattern index.

pascal OSErr PatternTableFrame0val (PatternTableRef, PatternIndex, Rect *r);

pascal OSErr PatternTableFillOval (PatternTableRef, PatternIndex, Rect *r)

Fills or frames the oval with the contents of the specified pattern index.

pascal OSErr PatternTableFramePoly (PatternTableRef, PattenIndex, PolyHandle);

pascal OSErr PatternTableFillPoly (PatternTableRef, PatternIndex, PolyHandle);

Fills or frames the polygon with the contents of the specified pattern index.

pascal OSErr PatternTableFrameRgn (PatternTableRef, PatternIndex, RgnHandle);

pascal OSErr PatternTableFillRgn (PatternTableRef, PatternIndex, RgnHandle);

Fills or frames the region with the contents of the specified pattern index.

Themes can also define new pattern types to take advantage of special theme-specific behavior, such as algorithmically defined patterns. To do this, the theme provides a resource defining the pattern type or registers a code fragment module or dynamically loaded library using, for example, the InstallPatternDefinition command described below. The pattern definition procedure will be added to the internal type list of the system, and will be called directly to load, unload and draw patterns of the corresponding type. This code can be stored as a code fragment module or dynamically loaded library, and remains loaded as long as there are pattern look-up table entries which reference its type. For this reason, pattern definitions can remain installed even after the theme which created the pattern is unloaded in case these definitions are used by other applications.

pascal OSErr InstallPatternDefinition (ConnectionID cfmnConnection);

Install the specified pattern definition in a pattern handler list. If a handler for that type has already been installed, an error is returned. The pattern definition's type (as returned by PatternDefGetType) should be unique, and the PatternDef is locked and loaded in the system heap.

When a pattern definition procedure is installed, it can be added to an internal pattern definition table. For speed, these pattern definition procedures can be referenced by index rather than type in the pattern look-up table. When a new pattern is added to the pattern look-up table, the pattern definition table is scanned and the index of the definition for the pattern type is inserted into the record for the new pattern. As new types are added, they can be added at the end of the list.

When a new pattern definition procedure is added to the internal pattern definition table, a list is built which includes the exported pointers contained in the pattern definition. If any standard pointers are not defined, they are set to a default pointer which simply returns an unimplemented error. As discussed above with reference to FIG. 9, when a pattern is drawn, the pattern is found in the pattern look-up table and its corresponding pattern definition procedure is located, then the desired finction pointer is called.

An example of a pattern definition procedure is shown below, which procedure is used to get a pattern defined on a per pixel basis from the look-up table and command the graphic subsystem to draw same.

```
    // structure for interpreting contents of our PatternData struct
typedef struct
    {
    PixPatHandle hPixPat;
    UInt32 unused;     // PatternData struct is 8 bytes, pad to fit
    }
PixPatData, *PixPatDataPtr;
OSErr PatDefOpen (OSType *pPatternType)
    {
    *pPatternType = 'ppat';    // return type
    *pRefCon = (RefCon) 0;    // no refcon used
    return noerr;
    }
OSErr PatDefClose ()
    {
    return noerr;
    }
OSErr PatDefLoadData (PixPatDataPtr *pdata, Int16 id, Int16 index)
    {
    pData->hPixPat = GetPixPat (id);
    if (pData->hPixPat == nil)
      return MemError();
    return noerr;
    }
OSErr PatDefFreeData (PixPatDataPtr *pdata)
    {
    DisposePixPat (pData->hPixPat);
    return noerr;
    }
OSErr PatDefSetData (PixPatDataPtr *pdata, PixPatDataPtr *pNewData)
    {
    if (!pData->hPixPat)
        {
        NewPixPat (&pData->hPixPat);
        if (!pData->hPixPat)
            return QDError ();
    CopyPixPat (pNewData->hPixPat, pData->hPixPat);
    return noerr;
    }
OSErr PatDefSetPen (PixPatDataPtr *pdata)
    {
    PenPixPat (pData->hPixPat);
    return noerr;
    }
OSErr PatDefFrameRect (PixPatDataPtr *pdata, Rect *prect)
    {
    FrameCRect (pData->hPixPat);
    return noerr;
    }
OSErr PatDefFillRect (PixPatDataPtr *pdata, Rect *prect)
    {
    FillCRect (pData->hPixPat);
    return noerr;
    }
```

The appearance management layer also defines a range of common pattern indices that can be included in each theme's pattern look-up table so that these indices are available to all clients. These include the set of patterns used to generate bevels and groups, along with other useful patterns, for example, the current background of the menu bar. The current background of the menu bar index may be passed to one of the standard theme drawing routines to draw shapes in the menu bar color. Below, for illustration purposes, an exemplary set of such common pattern indices is defined.

```
enum{
// standard beveling colors
kBevelBackgroundIndex = 0
kBevelFrameIndex,
kBevelFaceIndex,
kBevelShadowIndex,
kBevelHiliteIndex,
```

-continued

```
kBevelCornerIndex,
kBevelAuxShadowIndex,
kBevelAuxHiliteIndex,
kBevelAuxCornerIndex,
kBevelHiliteCorner,
kBevelShadowCorner,
kInvBevelFrameIndex,
kInvBevelFaceIndex,
kInvBevelShadowIndex,
kInvBevelCornerIndex,
kInvBevelHiliteIndex,
kInvBevelAuxShadowIndex,
kInvBevelAuxCornerIndex,
kInvBevelAuxHiliteIndex,
kInvBevelHiliteCorner,
kInvBevelShadowCorner,
// text frames
kTextFrameFillIndex,
kTextFrameFrameIndex,
kTextFrameHilightIndex,
kTextFrameShadowIndex,
// standard ridge and group indices
kGroupHilightIndex,
kGroupShadowIndex,
kGroupCornerIndex,
kGroupTextIndex,
kRidgeHilightIndex,
kRidgeShadowIndex,
kRidgeCornerIndex,
kRidgeAuxCornerIndex,
// beveled - shadowed text
kTextIndex,
kTextShadowIndex,
kTextHilightIndex,
kTextCornerIndex,
// custom
ThemeCustomRangeStart = 16384
};
```

typedef UInf16 ThemePatternIndex;

In addition to these exemplary defined types, a theme may define additional types which are used internally. These additional types can be indexed sequentially starting from whatever next highest index is available, e.g., 16384 in the example given above.

The following illustrates three exemplary pattern types which can be defined for usage in the appearance management layer. Therein, the command RGBColor specifies a red, blue or green color combination with which to draw an object or object part. ColorPattern describes a two-color 8×8 pixel pattern with a fore and backcolor, each specified with an RGBColor. An exemplary definition of a ColorPattern type is shown below:

```
typedef struct
{
RGBColor forecolor;
RGBColor backcolor;
Pattern pattern;
}
```

A PixPat type specifies an arbitrary pattern defined on a per-pixel basis, wherein a designated area may be filled or drawn with the pattern contents by the graphics subsystem. The PixPat (Pixel Pattern) data structure is defined by the graphics subsystem, and is used to contain this per-pixel pattern.

Themes provide a set of standard pattern look-up resources for use by the appearance management layer which are described below. The pattern lookup table defines the set of colors and patterns used by the theme and is used to build the theme's pattern look-up table. The part index table maps the set of standard theme pattern indices into the pattern look-up table. An exemplary implementation of a PatternLookupTable and a PartIndexTable is:

```
#define kPatRGBKind 'clut'          //color lookup table id + index
#define kPatPixKind 'ppat'          //PixPat id
#define kPatColorPatKind 'cpat'     //ColorPattern id
// Pattern Lookup Tables
typedef struct
    {
    OSType patternKind;             // kind of pattern, ie. kPatGBKind
    SInt16 patternID;               // pattern resource identifier
    UInt16 index;                   // index within resource
    UInt32 patternData [2];         // pattern data holder when loaded
    }
PatternLookupTableEntry;
typedef struct
    {
    UInt16 numEntries;              // count of entries in table
    PatternLookupTableEntry entries [];     // array of entries
    }
PatternLookupTable;
// Part Index Tables - maps a ThemePatternIndex into a Pattern Lookup Table
typedef struct
    {
    ThemePatternIndex index;    // corresponding ThemePatternIndex
    UInt16 plutIndex;           // PatternLookupTable index
    }
PartIndexEntry;
typedef struct
    {
    UInt16 numEntries;          // count of entries in table
    PartIndexEntry entries [];      // array of entries
    }
PartIndexTable;
```

As mentioned earlier, other exemplary embodiments of the present invention provide for pattern/color abstraction by returning information to the client rather than the appearance management layer commanding the graphic subsystem directly. According to these embodiments, the client will ask the appearance management layer for a structure, e.g., a PixPat structure, corresponding to a specified index and will receive a handle that will allow the application to make the appropriate drawing call to the graphic subsystem **56**. An example for this embodiment is illustrated below in pseudocode:

```
typedef struct
  {
  UInt32 data [2]; // data block for pattern definition use
  }
PatternData;
```

OSErr PattemnDefOpen (0);
Opens the pattern definition, initializing any global state data. The pattern def may return an error code to veto loading (for example if the pattern def cannot run on the current system configuration).
OSErr PatternDefClose (0);
Closes the pattern definition and frees any global state data. This is called prior to termination of the pattern def's connection.
OSErr PatternDefGetKind (OSType *pKind);
Returns the pattern kind identifier. This is invoked by the appearance management layer to find the link to entries in the pattern table.
OSErr PatternDefLoadData (PatternData *pData, SInt16 resId, UInt16 index);
Loads the pattern data from a resource, based upon the resource id+index.
OSErr PatternDefCloneData (PatternData *pData, PatData *pCopy);
Clones the pattern data contained in the PatData record and places the result in *pCopy.
OSErr PatternDefSetData (PatternData *pData, PatData *pNewData);
Sets the pattern data to a copy of that contained in *pNew-Data.
OSErr PatternDefUnloadData (PattemData *pData);
Frees the data stored in the pattern data record.
OSErr GetPatternPixPat (PatData *pData, PixPatHandle *hPixPat);
Returns the PixPat represented by the pattern data.
OSErr ApplyShapeStyle (PatData *pData, GXShape shape);
Modifies the state of the GX object so that it draws in the desired style. This may include creating ink, transform or other style objects and linking them to the shape, or modifying the shape itself.

Another example of how a client would interact with the pattern look-up tables **48** according to these exemplary embodiments is illustrated below.
OSErr NewPatternTable (PatternTable *table);
Creates a new pattern table.
OSErr GetNewPatternTable (SInt16 resID, PatternTable *table);
Gets a new pattern table from a resource.
OSErr DisposePatternTable (PatternTable *table);
Dispose a pattern table.
OSErr GetPatternDef (<PatternDef Reference>, SOMObject *patternDefObject);
Load a pattern definition proc and return its SOM Object.
OSErr AddPatternDefToTable (PatternTable table, SOMObject patternDefObject);
Add the pattern definition proc to the table.
OSErr PatternTableSetIndexData ( PatternTable table, UInt16 index, OSType kind, PatternData *pData);
Set the data record associated with the table index.
Application Pattern and Style Queries
OSErr ThemeGetPartPixPat (PartCode part, PixPatHandle *partPixPat);
Gets the PixPat associated with the part code.
OSErr ThemeApplyPartStyleToShape (PartCode part, GXShape shape);
Sets the style of the GXShape to the part style.
OSErr PatternTableGetPixPat ( PatternTable table, UInt16 index, PixPatHandle *hPixPat);
Gets the PixPat associated with the table+index.
OSErr PatternTableApplyStyleToShape ( PatternTable table, UInt16 index, GXShape shape);
Sets the style of the GXShape to that associated with the table+index.
OSErr ThemeGetPartSeed (UInt32 *seed);
Returns the seed for the theme pattern table. The seed is updated when changes are made to the pattern table which may invalidate cached PixPatsHandles.
OSErr PatternTableGetSeed (UInt32 *seed);
Returns the seed for the application pattern table. The seed is updated when changes are made to the pattern table which may invalidate cached PixPatsHandles.
SPI
OSErr InstallSystemPatDef (SOMObject patternDefObject);
Installs a pattern definition in the system PatDef table.

Having described two exemplary embodiments wherein pattern look-up tables can be used to abstract patterns and colors from the interface, another example is provided below in which both embodiments are applied to the exemplary application of filling a window rectangle with the bevel background color and then drawing a bevel inset two pixels in the window.

First, by way of the former, exemplary embodiment wherein drawing primitives are sent to the appearance management layer.

```
Rect bevelRect;
OSErr error;
GetWindowRect (&bevelRect);
// Fill window rectangle with bevel background
error=ThemeFillRect (kBevelBackground, &bevelRect);
// make bevel inset 2 pixels from window edge
InsetRect (&bevelRect, 2, 2);
// Draw Bevel on background
error=ThemeDrawBevel (&bevelRect, false);
```

Now, using the latter exemplary embodiment wherein the appearance management layer returns a data structure to the client.

```
Rect bevelRect;
OSErr error;
PixPatHandle hBackgroundPat;
GetWindowRect (&bevelRect);
// Get bevel background PixPat
error=ThemeGetPartPixPat (kBevelBackground, &hBackgroundPat);
// Fill window rectangle with bevel background
if (error==noErr) FillCRect (&bevelRect, hBackgroundPat);
// make bevel inset 2 pixels from window edge
InsetRect (&bevelRect, 2, 2);
// Draw Bevel on background
error=ThemeDrawBevel (&bevelRect, false);
```

Of course, those skilled in the art will appreciate that all of the pseudocode examples provided herein are intended to be exemplary and illustrative in nature.

Themes and Theme Switching

Having described exemplary systems and methods for abstracting the appearance and behavior of a user interface from its functionality using switchable drawing procedures and pattern look-up tables, the following description indicates how these capabilities can be used together to manifest sets of appearance and behavior attributes on a user interface which blend together to project a common theme. As described earlier, themes are coordinated designs of the interface elements which combine to create a distinct visual and audio environment on the display. According to one exemplary embodiment of the present invention, users can choose among different themes from, for example, an appearance control panel which can be activated on the desktop interface. An exemplary appearance control panel is illustrated as FIG. **11**.

In FIG. **11**, a pop-up, pull-down or drop-down menu **140** allows users to specify an overall appearance/behavior by selecting the theme to be installed. Beneath the theme setting box **140** to the left is an options area **142** in which a user may select various options within each theme. For example, a user could specify a background color, a font and a highlight color. To the right of the options area **142**, is a preview area **144** where exemplary interface elements of the theme currently selected in box **140** are shown so that a user can preview what the theme will look like before making a selection. Exemplary interface elements can include, for example, a desktop pattern, a menu bar and menu, an active window, and a dialog box with radio buttons, a checkbox, push buttons, and selected text. Using the appearance control panel, a user will be able to change the appearance of the desktop quickly and easily.

However, some users may desire even more control over the appearance and behavior of their desktop interface. Thus, according to another exemplary embodiment of the present invention, the appearance control panel can provide user selectibility over all of the objects which can be displayed on the user interface. For example, the appearance control panel could include a library of each type of interface object from which the user can select for inclusion in a user-defined theme. After selecting the different types of interface objects, the user can be prompted for a theme name under which pointers to the appropriate drawing procedures and other information for realizing the selected objects can be stored. According to still further exemplary embodiments of the present invention, an appearance object editor can be provided wherein a user can create his or her own interface objects using a library of parts provided by the object editor. For example, each of the glyphs illustrated in FIG. 5 can have a multitude of variations from which a user can create his or her own document window (both active and inactive). Once created, the new interface object can be stored in the library of interface objects from which user-defined themes can be created.

Theme attributes are a collection of theme properties that are both system-defined and theme-defined. Each of the theme's properties can be queried and set by appearance management layer functions. For example, the following properties can be defined by the system:

```
#define kThemeSystemFont                'sysf'
#define ThemeTextHighlightColor         'tcol'
```

To get a theme property, a get theme property function can be called for example by:

OSErr GetThemeProperty (OSType property, void *dataptr, Size dataSize)

This function will return the requested property from the current theme. If the current theme does not include the requested property, typeNotFoundErr is returned.

To set a theme property, call the SetThemeProperty function:

OSErr SetThemeProperty (OSType property, void *dataptr, Size datasize)

The SetThemeProperty command sets the specified theme property to the given data. Having described themes in general and ways in which themes can be created, selected and stored by a user, the following describes the operation of systems and methods according to the present invention once a theme change is requested by a user or an application beginning with FIG. **12**.

FIG. 12 illustrates interactions between, for example, a theme **70**, the appearance management layer **40**, and an application **38**. Therein block **48** includes the pattern tables as discussed above, and block **54** contains the animation and sound utilities which supplement the runtime routines of block **52**. Further, an icon **68** is shown which diagrammatically illustrates an appearance control panel **69**, e.g., the panel of FIG. **10**, which an end user can operate to switch themes.

A current theme's resource chain **72** is opened and managed by the theme switching **50** and runtime routines **52**. The resource chain **72** can include, for example, a theme attributes property list (e.g., behavior matrices as described above), theme preferences (e.g., a preferred background pattern, preferred system font, etc.), theme data resources (e.g., the pattern table which defines the set of patterns and colors used by the theme, pattern code procedures which allow definition of new pattern types, etc.) and override resources (e.g., icons for the theme which overrides system icons). The theme resource chain can be maintained separately from the resources of the currently running application, and can be switched in and out in response to a demand by either an application or a user (appearance control panel). The theme's resource chain **72** is setup whenever the appearance management layer **40** calls any of the theme's code.

As explained above with respect to the switchable drawing procedures according to exemplary embodiments of the present invention, when the appearance management layer is present, conventional drawing procedures (e.g., CDEF, LDEF, MDEF and WDEF) are replaced by the appearance management layer's switcher resources shown in FIG. **11** at blocks **74–80**. Externally, these switcher resources serve the identical function as traditional drawing procedures. Internally, they permit dynamic switching to the appropriate drawing procedures by the utilities when the theme changes. This switching can be accomplished by supplying new pointers **82–88** to the drawing procedures referenced by switcher resources **74–78**. In this way, when the switcher resources call back into the utilities as described above, the utilities will be pointed at the drawing procedures for the current theme.

The current theme is set by calling the appearance management layer's set theme function, for example., by the command:

OSErr SetTheme (const FSSpec *themefile)

The set theme function uses an FSSpec parameter that identifies the theme file that should be loaded and activated by the appearance management layer. In normal operation, this function loads the requested theme file, switches to the new theme and then releases the old theme. The old theme is released after the new theme is completely loaded so that

if the new theme could not be activated, the system can revert back to the original theme such that the user does not become stranded without an interface.

The exemplary steps illustrated in the flowchart of FIG. **13** can be executed to open the new theme file. At block **100**, a new theme info record is created. This data structure contains all of the global information that the appearance management layer uses to keep track of the state of the current theme and contains its own resource chain information, e.g., procedure pointer tables for the switcher, the theme property list, the theme pattern tables, etc.

Next, the appearance management layer creates a new resource chain at block **102**. The new theme's resource file is then opened at **104** after which the theme's runtime code is loaded, at **106**, and its open function is called. At this time, the new theme can test the operating conditions of the system to determine if the load should continue or be aborted. If the load aborts, the theme may present an alert to the user as to why the theme could not be loaded. If the theme has its own preferences file, it can be opened by the theme at this time.

The theme's property list is loaded at block **108**, for example, by calling a get resource function. This allows the property list to come from any preferences file that may have been opened in the previous step. If a property list is found, it is stored in the theme info record. Subsequently, at block **110**, the theme's pattern look-up table is loaded. First, all pattern definition procedure resources are loaded. Then the standard pattern look-up table and part index table resources are loaded. The pattern look-up table is then built from the contents of these resources. A pointer table to be used by the switcher resources is then built as shown by block **112**. This table is stored in the theme info record. Lastly, the new theme's initialize function is called at block **114**. The new theme can allocate memory or load extra resources that it requires while being active.

FIG. **14** illustrates steps that can be executed to switch from an old theme to a new theme. First, a transition effect can be presented as block **116**. For example, the screen may fade to black, a dialog can be presented, or the themes could gradually blend from one to the other, e.g., "morphing". Then, the old theme's resource chain is switched in as described by block **118**. All of the drawing procedures are called with a deallocate message. These messages **120** are sent to the appearance management layer's switcher definition procedures, which are currently routing messages to the old theme's implementations of the definition procedures. This allows any of the theme's definition functions to deallocate any global data that they may have been allocated.

The appearance management layer sets the new theme info record as the current theme's information record at **122**. Once the new theme info record is set, all of the external calls into the appearance management layer will affect the new theme. The new theme's resource chain is switched in at block **124**. All of the drawing procedures are called with an initialize message. These messages are sent to the appearance management layer's switcher resources, which are currently routing messages to the new theme's implementations of the drawing procedures. This allows any of the theme's definition functions to allocate any global data that they may need.

The steps executed to release the old theme file are shown in FIG. **15**. First, at block **128**, the old theme's resource chain is switched in. Next, the old theme's deallocate function is called at **130**. The theme is responsible for disposing of any allocations that it may have made when it received its initialize message. The old pointer table used by the switcher definition procedures is disposed of per block **132**. Then, the old theme's pattern look-up table and property list are disposed of as denoted by blocks **134** and **136**, respectively. The files in the old theme's resource chain can then be closed and the resource chain disposed of prior to disposing of the old theme's theme info record (blocks **138** and **140**).

If an error occurs while trying to open and load the new theme or while switching from the old theme to the new theme, the switch is aborted and the set theme function attempts to reverse all of the steps that have already successfully completed so that the system continues to generate an interface using the old theme. The error that caused the switch to abort can be returned by this function. To request that the default system theme is switched in, either an FSSpec parameter to the system file or NIL can be passed in the themefile parameter.

To determine what theme file is currently active, a get theme function can be called, for example by the command:

OSErr GetTheme (FSSpec *currentThemeSpec)

An FSSpec parameter value referencing the currently active theme file will be returned in the currentThemeSpec parameter. If the current theme is the default system theme, an FSSpec referencing the system file will be returned. If an error occurs while attempting to locate the FSSpec of the current theme, an appropriate error code will be returned and the currentThemeSpec parameter will remain unchanged.

Normally, the current theme's resource file is not present in the currently running application's resource chain. This can be done to prevent resource identification conflicts between applications, the operating system and the current theme. The appearance management layer maintains a separate resource chain that contains the current theme file and any other files that the current theme may have opened (such as a preferences file). When the appearance management layer executes code in the theme, the theme's resource chain is setup by the appearance management layer, which allows for normal GetResource calls to be used to get theme resources. If an application wishes to gain access to the current theme's resources, several functions can be provided. For example, to get a resource from the current theme file, a get theme resource function can be called, for example:

Handle GetThemeResource (OSType restype, UInt16 id)

GetThemeResource has the same function as the GetResource function, except that this command gets the resource from the current theme's resource chain.

If more flexibility is needed when getting resources from the current theme file, the low-level appearance management layer function GetThemeTopMapHandle may be used to get the top of the current theme's resource chain.

OSErr GetThemeTopMapHandle (Handle *themeMap)

The GetThemeTopMapHandle function returns the top map handle that contains the current theme file and any other opened theme files (such as a preferences file) and all of the system resource maps. Caution should be exercised when using the GetThemeTopMapHandle function to avoid leaving the theme's resource chain switched in when executing utility functions or after returning to other parts of an application's code. When the theme's resource chain is switched in, the application's resource chain is unavailable. Note also that when the theme changes, this map handle and associated resources will no longer be valid, so this value should not be cached.

A theme can implement three theme definition functions that the appearance management layer calls when a theme is

being loaded or disposed of. When the appearance management layer begins to switch to a theme, immediately following that theme's resource file being opened, the theme's function can be called.

pascal OSErr ThemeFilePreflight (void *themedata)

The theme's test function is called before any resources are loaded by the appearance management layer. In this way, the theme has an opportunity to test the conditions of the operating system (such as memory or graphics capability). If the test function returns an error, the appearance management layer will close the theme file and not attempt to continue loading. If the test function returns no error, the appearance management layer continues to load the theme, as described above.

The themedata parameter returned by the exemplary test function shown above is used by the theme to allocate and store any global data that the theme wishes to keep for itself. On entry to the test function, the themedata parameter points to NIL. The test function (or any of the other theme definition functions) may change the value pointed to by themedata. This themedata value is persistent as long as the theme remains loaded.

When the appearance management layer is finished loading all of the theme's resources and loading each of the theme's standard definition procedures, the theme's initialize function is called, for example:

pascal OSErr ThemeFileInitialize (void *themedata)

The theme's initialize function can be used to do any special processing after the appearance management layer has completely loaded the theme. It may allocate data structures, load additional resources, open preferences files, setup its theme property list, etc. The themedata parameter points to a global storage location useful for storing a pointer to the themes global data. If the theme's initialize function returns an error, the appearance management layer will abort the switch to the theme. The appearance management layer will dispose of any allocations it has already made and close the theme file.

When the appearance management layer is preparing to unload a theme, the theme's dispose function is called, for example:

pascal OSErr ThemeFileDispose (void *themedata)

The dispose function should dispose of any allocations that were made with either the test or initialize functions. The theme file then has an opportunity to store any resources in its preferences file and/or set its theme properties. After the theme returns from this function, the appearance management layer will deallocate all of the appearance management layer's storage for the theme and close the theme's file.

Those skilled in the art will appreciate that the foregoing described exemplary embodiments can be implemented using, for example, various types of computer systems. A typical computer system can have a monitor or display connected to a processor for display of the graphical user interfaces described herein. The computer system can also have known I/O devices (e.g., CD drives, floppy disk drives, hard drives, etc.) which can store and read programs and data structures used to implement the above-described techniques. These programs and data structures can be encoded on such computer-readable media. For example, the layout resources described above can be stored on computer-readable media independently of the computer-readable medium on which the layout engine itself resides.

The above-described exemplary embodiments are intended to be illustrative in all respects, rather than restrictive, of the present invention. Thus the present invention is capable of many variations in detailed implementation that can be derived from the description contained herein by a person skilled in the art. All such variations and modifications are considered to be within the scope and spirit of the present invention as defined by the following claims.

What is claimed is:

1. In a graphical user interface, a method for drawing objects comprising the steps of:

receiving, by a first layer, a command from an application to draw an object on said user interface;

translating, by said first layer, said command into a drawing primitive which includes a pointer to data associated with said object;

sending said drawing primitive to a layout engine;

retrieving, by said layout engine, said data associated with said object; and

drawing, by said layout engine, said object on said user interface.

2. The method of claim 1, wherein said first layer is a window definition object and said object is a window.

3. The method of claim 1, wherein said first layer is a menu definition object and said object is a menu.

4. A computer readable medium encoded with instructions comprising:

a data driven layout engine for drawing an object on a graphical user interface, the layout engine receiving a command to draw said object wherein said command includes a pointer to a layout resource and the layout resource includes a list of boundary and part information that is traversed only once by said layout engine to draw said object.

5. The computer readable medium of claim 4, wherein said object is a window.

6. The computer readable medium of claim 4, wherein said object is a menu.

7. The computer readable medium of claim 4, wherein said layout resource comprises a data structure including:

a list of parameters including:

a first sublist of parameters associated with boundaries of said object; and

a second sublist of parameters associated with parts of said object.

8. The computer readable medium of claim 7, wherein said first sublist of parameters includes a data structure having a value that indicates a number of boundaries defined within said layout resource.

9. The computer readable medium of claim 7, wherein said first sublist of parameters includes at least one boundary data structure which defines a boundary associated with said object, said boundary data structure including a start boundary parameter which identifies a first previously placed boundary relative to which said boundary is to be placed on said user interface.

10. The computer readable medium claim 9, wherein said boundary data structure further includes an offset parameter having a value which is used to determine a relative positioning of said boundary to said previously placed boundary.

11. The computer readable medium of claim 10, wherein said boundary data structure further includes an offsettype parameter having a value which indicates a manner in which said offset parameter is used to determine a relative positioning of said boundary to said previously placed boundary.

12. The computer readable medium of claim 11, wherein if said offsettype parameter equals a predetermined parameter then said offset parameter is added to said previously placed boundary to obtain said boundary's location.

13. The computer readable medium of claim 11, wherein if said offsettype parameter equals a predetermined parameter then said offset parameter is used as an index into an array and a value stored at said array index is added to said previously placed boundary to obtain said boundary's location.

14. The computer readable medium of claim 9, said boundary data structure includes a limit boundary parameter which identifies a second previously placed boundary relative to which said boundary is to be placed on said user interface.

15. The computer readable medium of claim 14, wherein said at least one boundary data structure farther includes a limittype parameter which constrains placement of said boundary relative to said start boundary.

16. The computer readable medium of claim 15, wherein if said limittype parameter is set to a predetermined parameter and said boundary is to be placed beyond a threshold, then said boundary is instead placed at said limit boundary.

17. The computer system of claim 14, wherein if said limittype parameter is set to a predetermined parameter and said boundary is to be placed beyond a threshold, then said boundary is instead placed at said start boundary.

18. The computer readable medium of claim 7, wherein if said offsettype parameter equals a predetermined parameter then said offset parameter is ignored and said boundary is placed halfway between said previously placed boundary and a limit boundary.

19. The computer readable medium of claim 7, wherein said first sublist includes a first set of data structures associated with boundary placement relative to one or more existing boundaries and a second set of data structures associated with boundary limits.

20. The computer readable medium of claim 7, wherein said second sublist of parameters includes a data structure having a value that indicates a number of parts defined within said layout resource.

21. The computer readable medium of claim 7, wherein said second sublist of parameters includes at least one part data structure which defines a part associated with said object, said part data structure including an inclusion attribute mask which specifies drawing of said part based upon a presence of at least one attribute.

22. The computer readable medium of claim 21, wherein said part data structure further includes an exclusion attribute mask which specifies drawing of said part based upon an absence of at least one attribute.

23. The computer readable medium of claim 22, wherein said inclusion attribute mask and said exclusion attribute mask both have bits associated with a same attribute.

24. The computer readable medium of claim 23, wherein, if neither of said associated bits in said inclusion attribute mask and said exclusion attribute mask are set, then said part will be drawn regardless of whether said object includes said same attribute.

25. The computer readable medium of claim 23, wherein if said associated bit of said inclusion attribute mask is set and said associated bit of said exclusion attribute mask is not set, then said part will be drawn only if said object has said same attribute.

26. The computer readable medium of claim 23, wherein if said associated bit of said inclusion attribute mask is not set and said associated bit of said exclusion attribute mask is set, then said part will be drawn only if said object does not have said same attribute.

27. The computer readable medium of claim 23, wherein if said associated bit of said inclusion attribute mask is not set and said associated bit of said exclusion attribute mask is not set, then said part will be drawn regardless of whether said object has said same attribute.

28. The computer readable medium of claim 7, wherein said second sublist of parameters includes at least one part data structure which defines a part associated with said object, said part data structure including a regionset parameter which identifies at least one set to which said part belongs.

29. The computer readable medium of claim 7, wherein said second sublist of parameters includes at least one part data structure which defines a part associated with said object, said part data structure including a plurality of parameters which identify boundaries within which said part is to be drawn.

30. The computer readable medium of claim 7, wherein said second sublist of parameters includes at least one part data structure which defines a part associated with said object, said part data structure including at least one parameter which identifies a manner in which said part is to be filled.

31. The computer readable medium of claim 30, wherein said at least one parameter identifies a color pattern to be used to fill said part.

32. The computer readable medium of claim 31, said part data structure further includes a parameter which identifies an orientation for said color pattern relative to said part.

* * * * *

UNITED STATES PATENT AND TRADEMARK OFFICE

# CERTIFICATE OF CORRECTION

PATENT NO. : 6,243,102 B1
DATED : June 5, 2001
INVENTOR(S) : Joseph Ruff et al.



It is certified that error appears in the above-identified patent and that said Letters Patent is hereby corrected as shown below:

Title page,
**Related U.S. Application Data**, line 2, delete the word "continuation", insert -- continuation-in-part --.

Signed and Sealed this

Second Day of April, 2002

*Attest:*

*Attesting Officer*

JAMES E. ROGAN
*Director of the United States Patent and Trademark Office*

US006239798B1

(12) **United States Patent**
**Ludolph et al.**

(10) **Patent No.: US 6,239,798 B1**
(45) **Date of Patent: *May 29, 2001**

(54) **METHODS AND APPARATUS FOR A WINDOW ACCESS PANEL**

(75) Inventors: **Frank E. Ludolph**, Menlo Park; **George Tharakan**, Sunnyvale, both of CA (US)

(73) Assignee: **Sun Microsystems, Inc.**, Palo Alto, CA (US)

( * ) Notice: This patent issued on a continued prosecution application filed under 37 CFR 1.53(d), and is subject to the twenty year patent term provisions of 35 U.S.C. 154(a)(2).

Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 0 days.

(21) Appl. No.: **09/085,456**

(22) Filed: **May 28, 1998**

(51) **Int. Cl.**[7] ..... **G06F 3/14**

(52) **U.S. Cl.** ..... **345/340**; 345/339; 345/348; 345/342; 345/357

(58) **Field of Search** ..... 345/342, 339, 345/348, 340, 357, 358

(56) **References Cited**

U.S. PATENT DOCUMENTS

| | | | | |
|---|---|---|---|---|
| 5,072,412 | | 12/1991 | Henderson, Jr. et al. . | |
| 5,233,687 | | 8/1993 | Henderson, Jr. et al. . | |
| 5,394,521 | | 2/1995 | Henderson, Jr. et al. . | |
| 5,488,686 | * | 1/1996 | Murphy et al. | 345/357 |
| 5,533,183 | * | 7/1996 | Henderson, Jr. et al. | 345/344 |
| 5,745,096 | | 4/1998 | Ludolph et al. | 345/340 |
| 5,757,371 | * | 5/1998 | Oran et al. | 345/348 |
| 5,825,348 | * | 7/1998 | Ludolph et al. | 345/342 |
| 5,874,958 | * | 2/1999 | Ludolph | 345/339 |
| 5,920,316 | * | 7/1999 | Oran et al. | 345/348 |
| 5,923,326 | * | 7/1999 | Bittinger et al. | 345/340 |

OTHER PUBLICATIONS

Henderson, Jr. et al., "Rooms: The Use of Multiple Virtual Workspaces to Reduce Space Contention in a Window-Based Graphical User Interface",ACM transactions on Graphics, vol. 5, No. 3, Jul. 1986, pp. 211–243.

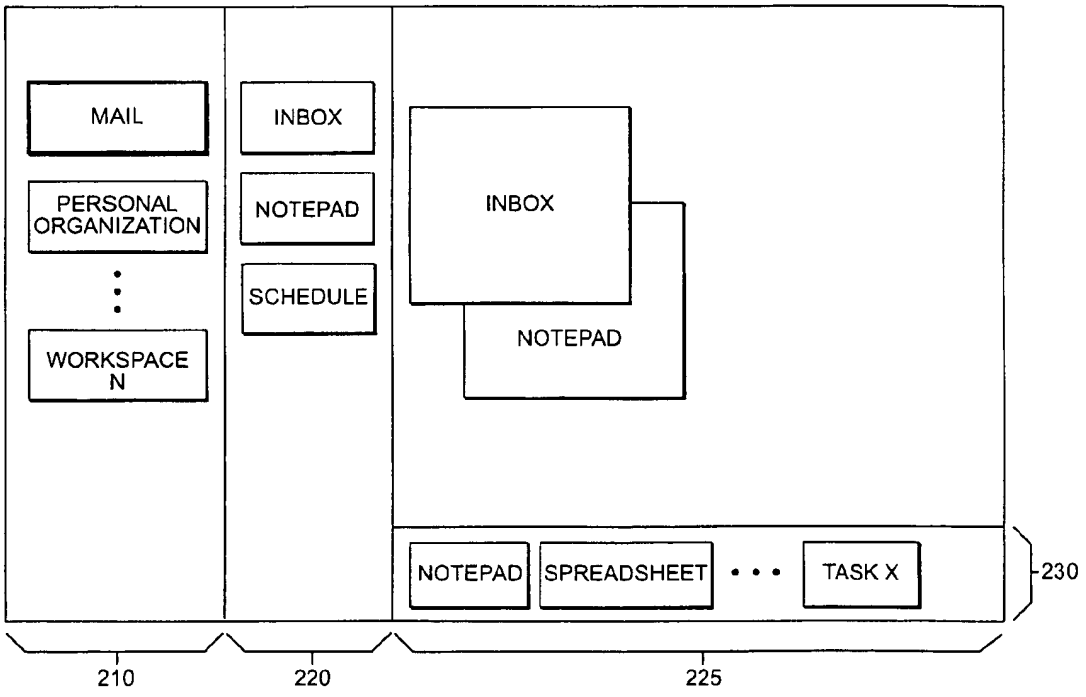
* cited by examiner

*Primary Examiner*—Raymond J. Bayerl
*Assistant Examiner*—Thomas T Nguyen
(74) *Attorney, Agent, or Firm*—Finnegan, Henderson, Farabow, Garrett & Dunner, L.L.P.

(57) **ABSTRACT**

A system employs a sliding window panel that contains icons representing every task that has been opened into a window, regardless of the workspace in which it exists. A user may use the sliding window panel to launch, terminate, hide, or resize windows in the workspaces.

**47 Claims, 8 Drawing Sheets**



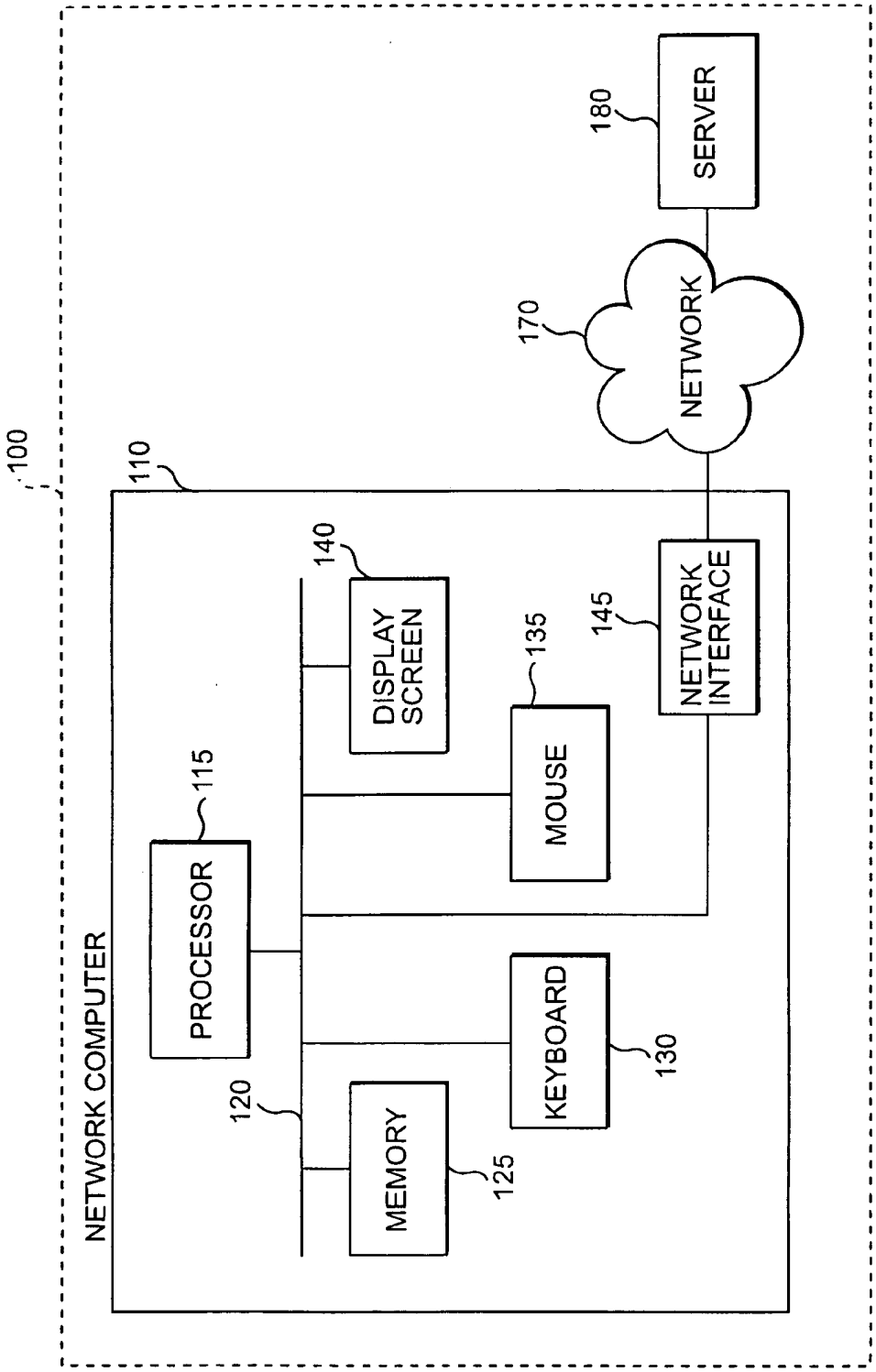

100
110
NETWORK COMPUTER
PROCESSOR
115
120
MEMORY
125
KEYBOARD
130
MOUSE
135
DISPLAY SCREEN
140
NETWORK INTERFACE
145
NETWORK
170
SERVER
180

FIG. 1

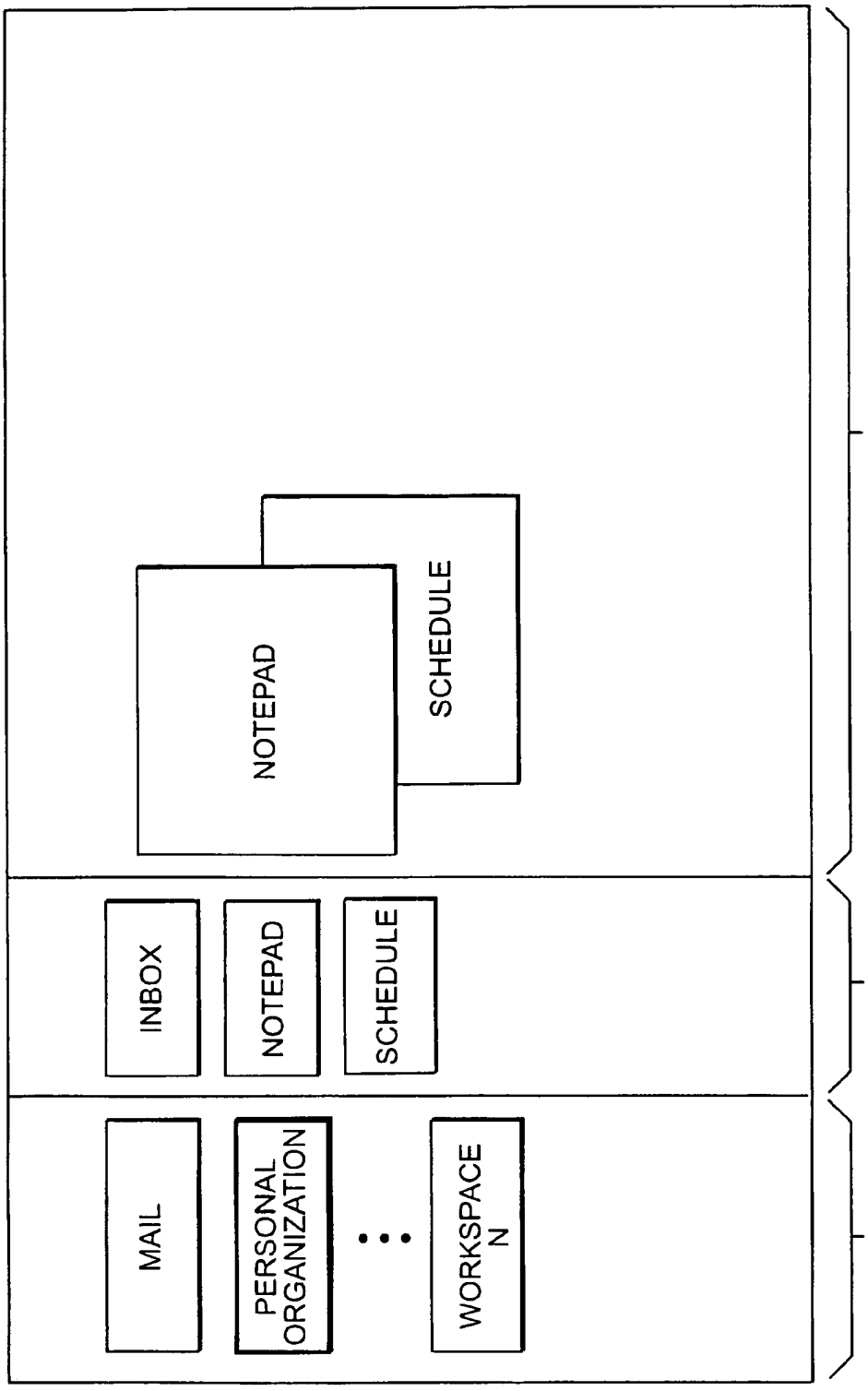

NOTEPAD
SCHEDULE
INBOX
NOTEPAD
SCHEDULE
MAIL
PERSONAL ORGANIZATION
...
WORKSPACE N
225
220
210

FIG. 2a

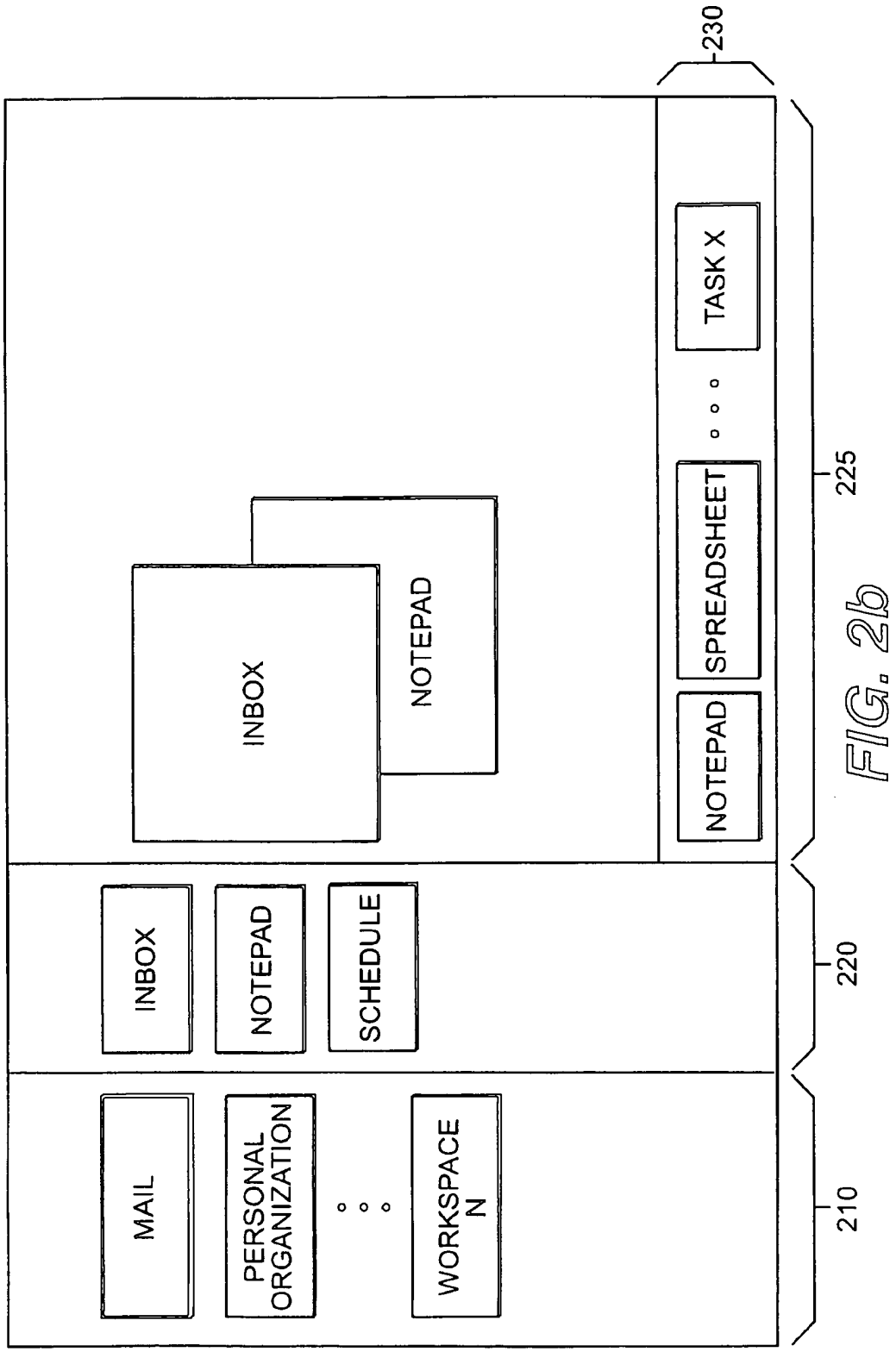

230
225
220
210
MAIL
PERSONAL ORGANIZATION
WORKSPACE N
INBOX
NOTEPAD
SCHEDULE
INBOX
NOTEPAD
NOTEPAD
SPREADSHEET
TASK X

FIG. 2b

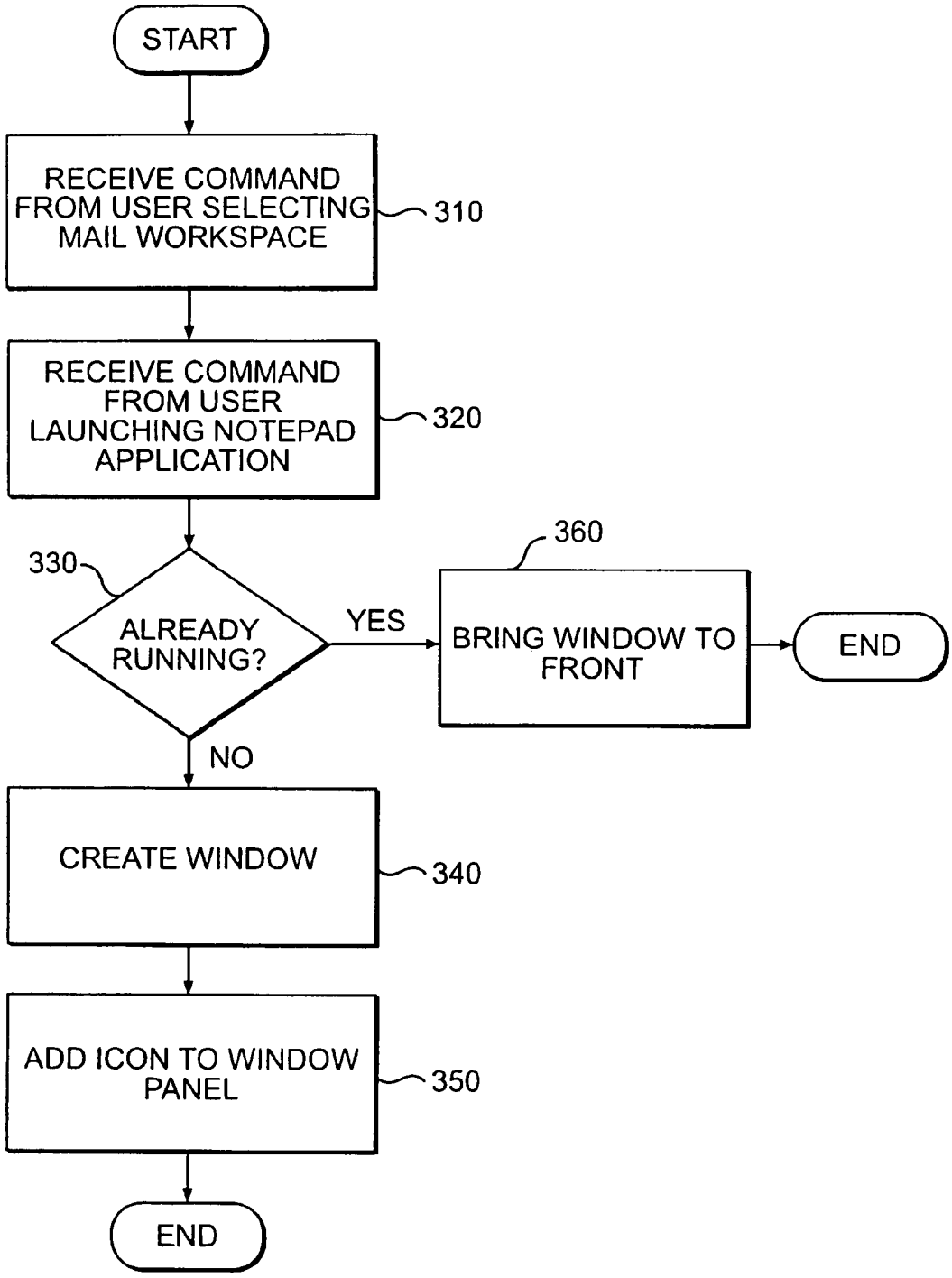

START
RECEIVE COMMAND FROM USER SELECTING MAIL WORKSPACE
310
RECEIVE COMMAND FROM USER LAUNCHING NOTEPAD APPLICATION
320
330
ALREADY RUNNING?
YES
360
BRING WINDOW TO FRONT
END
NO
CREATE WINDOW
340
ADD ICON TO WINDOW PANEL
350
END

FIG. 3

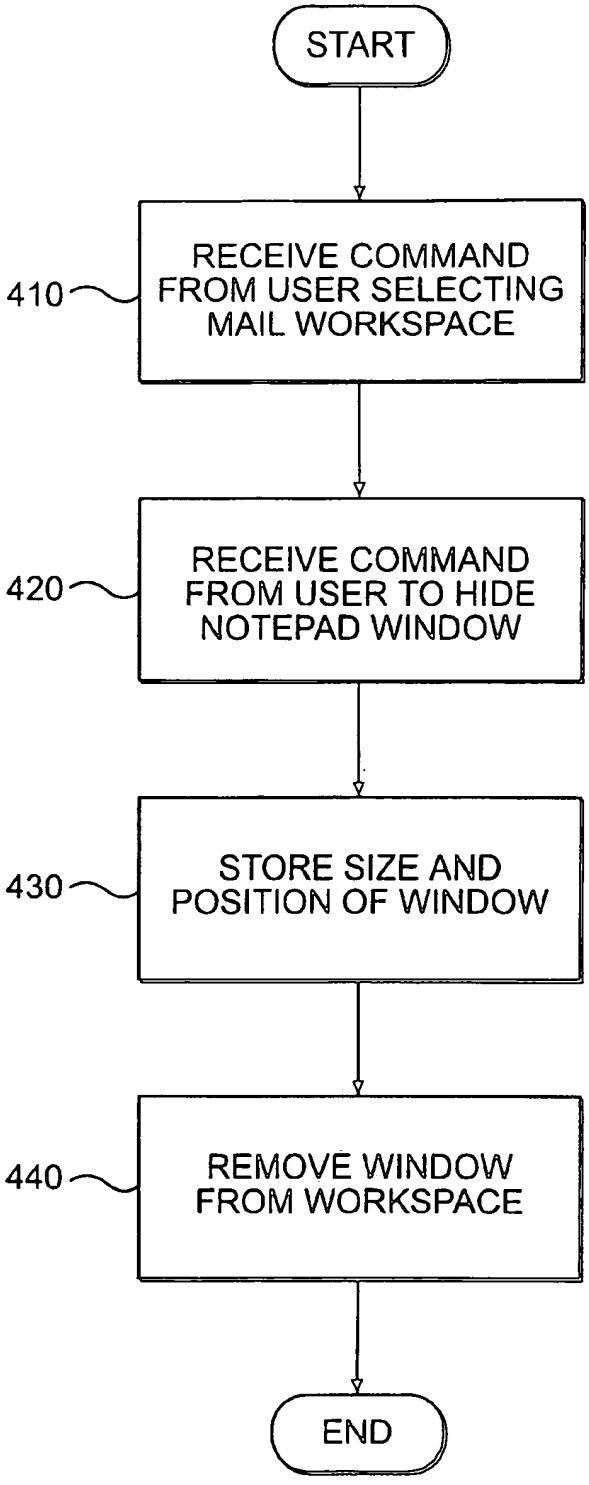

START
410
RECEIVE COMMAND FROM USER SELECTING MAIL WORKSPACE
420
RECEIVE COMMAND FROM USER TO HIDE NOTEPAD WINDOW
430
STORE SIZE AND POSITION OF WINDOW
440
REMOVE WINDOW FROM WORKSPACE
END

FIG. 4

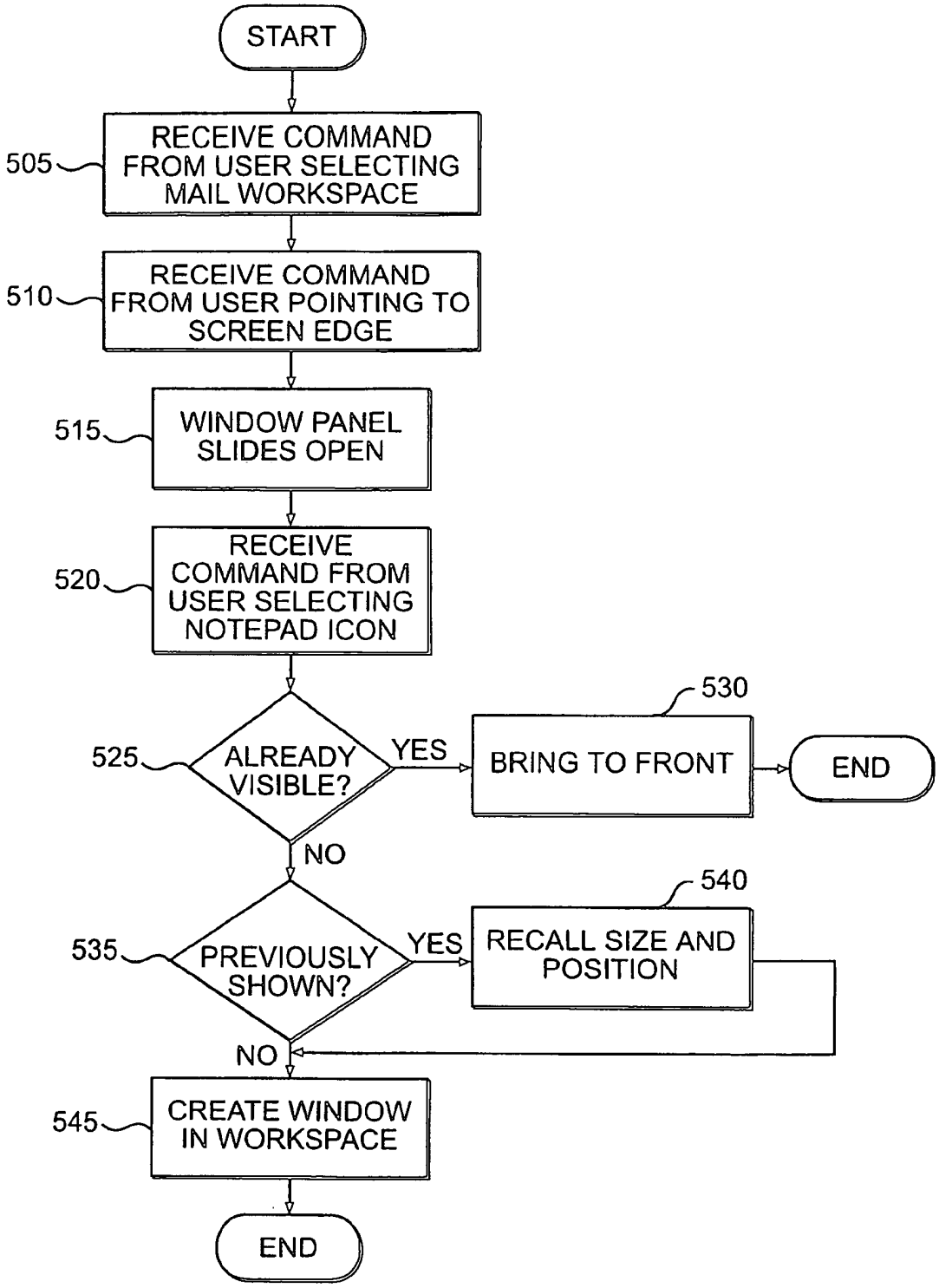

START
505
RECEIVE COMMAND FROM USER SELECTING MAIL WORKSPACE
510
RECEIVE COMMAND FROM USER POINTING TO SCREEN EDGE
515
WINDOW PANEL SLIDES OPEN
520
RECEIVE COMMAND FROM USER SELECTING NOTEPAD ICON
525
ALREADY VISIBLE?
YES
530
BRING TO FRONT
END
NO
535
PREVIOUSLY SHOWN?
YES
540
RECALL SIZE AND POSITION
NO
545
CREATE WINDOW IN WORKSPACE
END

FIG. 5

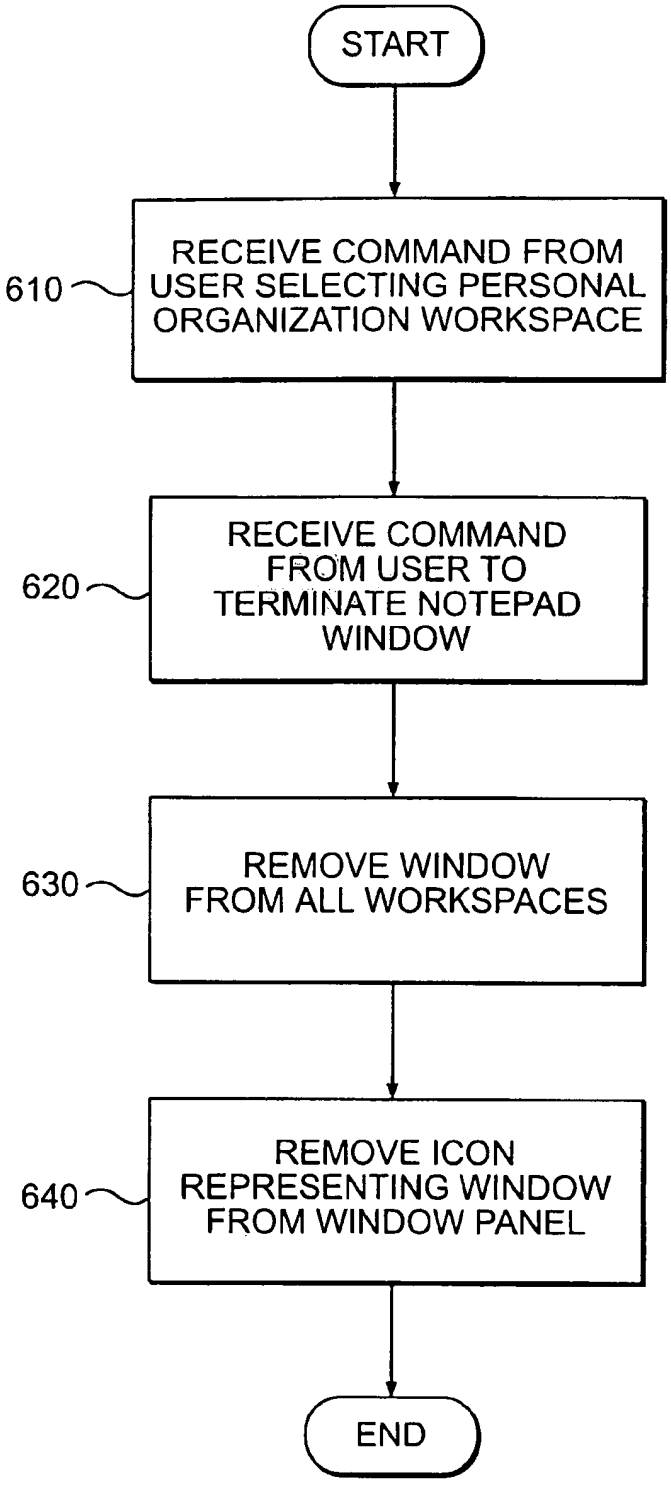

START
610
RECEIVE COMMAND FROM USER SELECTING PERSONAL ORGANIZATION WORKSPACE
620
RECEIVE COMMAND FROM USER TO TERMINATE NOTEPAD WINDOW
630
REMOVE WINDOW FROM ALL WORKSPACES
640
REMOVE ICON REPRESENTING WINDOW FROM WINDOW PANEL
END

FIG. 6

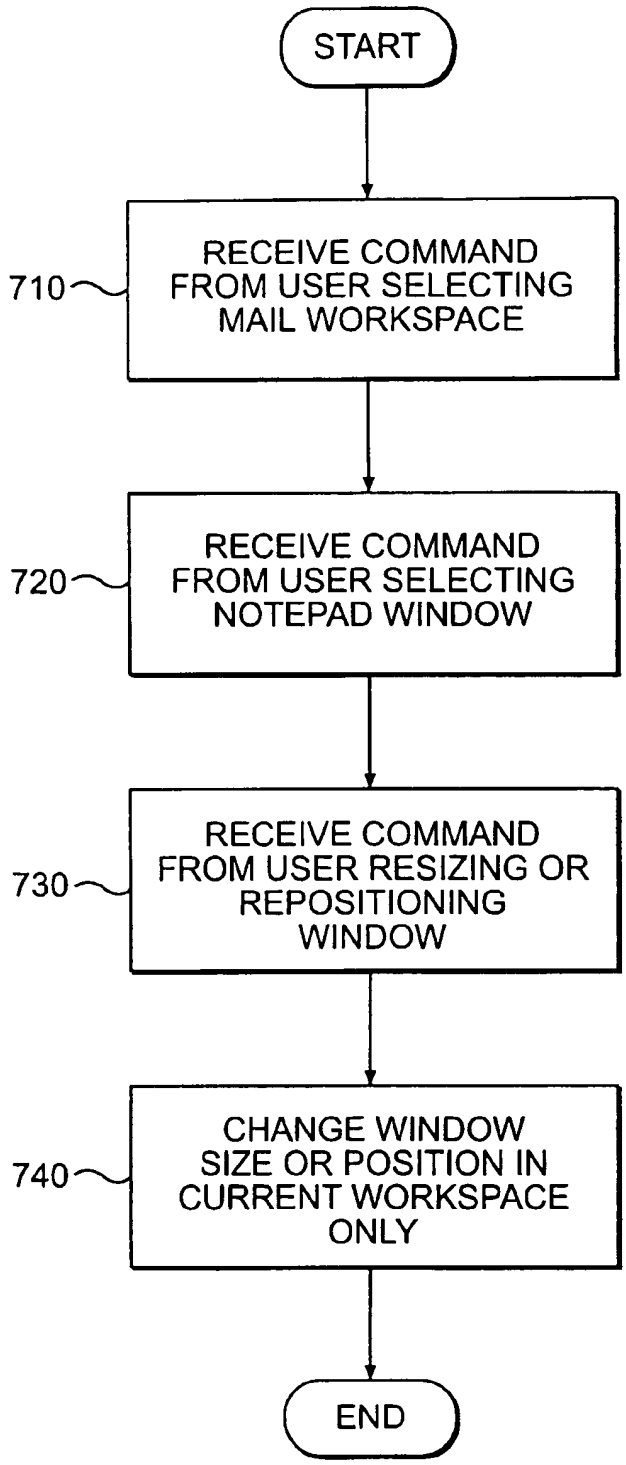

START
710
RECEIVE COMMAND FROM USER SELECTING MAIL WORKSPACE
720
RECEIVE COMMAND FROM USER SELECTING NOTEPAD WINDOW
730
RECEIVE COMMAND FROM USER RESIZING OR REPOSITIONING WINDOW
740
CHANGE WINDOW SIZE OR POSITION IN CURRENT WORKSPACE ONLY
END

FIG. 7

# METHODS AND APPARATUS FOR A WINDOW ACCESS PANEL

## BACKGROUND OF THE INVENTION

A. Field of the Invention

This invention relates generally to graphic user interfaces for computer systems and, more particularly, to methods and apparatus for providing a window access panel.

B. Description of the Related Art

Many modem computer systems employ graphic user interfaces that provide a separate "window" for each active task (as used herein, the term "task" includes both application programs and documents). Familiar examples include the Macintosh user interface from Apple Computer, Inc., and the Windows 95 user interface from Microsoft Corp. Such user interfaces facilitate computing because they provide a convenient way for a user to manage multiple tasks that are concurrently executing on the same computer.

Some user interfaces take this concept a step further by providing support for multiple workspaces. An example of such a user interface is the Unix Common Desktop Environment (CDE), which is based upon the Hewlett-Packard Visual User Environment. CDE was developed jointly by a group of companies including Sun Microsystems, Inc., Hewlett-Packard, and International Business Machines. A workspace is a collection of tasks that are employed to accomplish a specific objective. For example, a user may have one workspace for creating a newsletter and another for personal organization. The newsletter workspace may contain windows for numerous tasks, including word processing, drawing, and desktop publishing. Similarly, the personal organization workspace may contain windows for tasks that provide calendaring, scheduling, and addressing capabilities. A user of an interface that supports multiple workspaces can switch between those workspaces as needed, and the computer displays windows belonging to the selected workspace.

This capability of having multiple windows, while elegant in many respects, becomes somewhat cumbersome when the number of windows becomes large. Having numerous windows leads to a cluttered desktop appearance and makes it difficult for the user to locate a specific window corresponding to a task. As used herein, the term "task" may include software executing using a computer processor. Depending on the size of the display screen and on the task actively manipulated by the user, some of the windows may be completely visible while others may be partially visible or completely hidden.

Additional problems arise in user interfaces that support multiple workspaces. For example, a user may desire to display the same window in more than one workspace. In doing so, the user may desire to have a window in one workspace located at a different screen position, or sized differently than, the corresponding window in another workspace.

These problems have been addressed somewhat by others. For example, Microsoft has created a "Taskbar" for the Windows 95 operating system that typically resides at the bottom of the user's screen. A user may configure the Taskbar to reside out of view, normally, and to slide open when the mouse pointer touches the edge of the screen. Whenever a user launches a task, a window for that task is opened on the display (also referred to as a desktop), and a button for that task is added to the Taskbar. A Windows 95 user can choose to "hide" a task by selecting a button in the top right corner of the corresponding window. This causes the computer to remove the corresponding window from the desktop, but the button for the task remains in the Taskbar. The user can also "show" a previously hidden task by using a mouse to point at a button in the Taskbar and clicking the mouse button. This causes the computer to display the corresponding window on the desktop. The user can also "close" a task by selecting a button in the top right corner of the task's window. This causes the computer to terminate execution of the program, remove the corresponding window from the desktop, and remove the task's icon from the Taskbar.



Although the Taskbar provides significant window manipulation capabilities, it does not address the problem of window management in a user interface that supports multiple workspaces. This is primarily because Windows 95, itself, does not support multiple workspaces.

CDE provides a menu and dialog mechanism for managing window displays in multiple workspaces, but this mechanism has several deficiencies. For example, when a CDE user launches a task in a particular workspace, a corresponding window is opened in that workspace only. If the CDE user desires to view that particular window in another workspace, the user must remember which workspace contains the desired window, switch to that workspace, use a menu and dialog to specify the first workspace, and then switch back to the first workspace. Moreover, the window appears in the same screen position and at the same size in each workspace; any change in screen position or size in one workspace affects the screen position and size in all other workspaces.

There is therefore a need for a system that alleviates these problems and allows a user to easily select and manipulate windows in a user interface that supports multiple workspaces.

## SUMMARY OF THE INVENTION

Systems and methods consistent with the present invention, as embodied and broadly described herein, manage the display of windows corresponding to tasks executable by a computer. Multiple workspaces are provided, each of which is capable of displaying multiple windows corresponding to executing tasks. A window panel may be displayed that includes icons corresponding to the executing tasks, and permits shared access to a window of the executing tasks upon selection of a corresponding icon displayed in the window panel.

In accordance with the invention, a computer-readable medium contains instructions for managing the display of windows corresponding to tasks executable by a computer. This is accomplished by providing multiple workspaces, each of which is capable of displaying multiple windows corresponding to executing tasks. A window panel may be displayed that includes icons corresponding to the executing tasks, and permits shared access to a window of the executing tasks upon selection of a corresponding icon displayed in the window panel.

In accordance with the invention, a system for managing windows in a user interface that supports multiple workspaces comprises a memory having program instructions, and a processor. The processor is configured to use the program instructions to providing multiple workspaces, each of which is capable of displaying multiple windows corresponding to executing tasks. The processor is also configured to display a window panel that includes icons corresponding to the executing tasks, and to permit shared access

to a window of the executing tasks upon selection of a corresponding icon displayed in the window panel.

## BRIEF DESCRIPTION OF THE DRAWINGS

The accompanying drawings, which are incorporated in and constitute a part of this specification, illustrate an embodiment of the invention and, together with the description, serve to explain the advantages and principles of the invention. In the drawings, FIG. **1** is a block diagram of a computer system in which systems consistent with the present invention may be implemented;

FIG. **2***a* is a representative user interface screen showing a workspace consistent with the present invention;

FIG. **2***b* is a representative user interface screen showing another workspace consistent with the present invention;

FIG. **3** is a flow diagram of operations performed to launch a task consistent with the present invention;

FIG. **4** is a flow diagram of operations performed to hide a window consistent with the present invention;

FIG. **5** is a flow diagram of operations performed to show a hidden window consistent with the present invention;

FIG. **6** is a flow diagram of operations performed to close a window consistent with the present invention; and

FIG. **7** is a flow diagram of operations performed to reposition or resize a window consistent with the present invention.

## DETAILED DESCRIPTION

Reference will now be made in detail to an implementation of the present invention as illustrated in the accompanying drawings. Wherever possible, the same reference numbers will be used throughout the drawings and the following description to refer to the same or like parts.

A. Overview

Systems and methods consistent with the present invention operate in a graphic user interface that supports multiple workspaces. The systems and methods employ a sliding window panel that contains icons representing every task opened into a window, regardless of the workspace in which it exists. Whenever a new task is opened into a window, an icon representing that task is added to the window panel.

A user may hide a window in one workspace, but that window remains visible in any other workspaces in which it existed. The computer retains an icon representing the hidden window in the window panel so that the window may be subsequently shown. Moreover, the computer stores the location and size of the window so that when it is subsequently shown, it may be displayed as it existed when it was hidden. Alternatively, a user may close a window that is no longer needed. The window is removed from every workspace in which it existed, and the corresponding icon is removed from the window panel.

A user may show a window by selecting its icon from the window panel. If that window is already open in the workspace but obscured by other windows in front of it, it is simply brought to the foreground of the display. If the window was hidden, it is restored in the workspace at the same position and at the same size as it existed before it was hidden. If the window has never been shown in the workspace, it is shown at a default location and size. A user may also reposition or resize a window in a workspace, without affecting the size or position of the corresponding window in another workspace.

B. Architecture

FIG. **1** is a block diagram of a computer system **100** in which systems consistent with the present invention may be implemented. System **100** consists of a computer **110** connected to a server **180** via a network **170**. Network **170** may be a local area network (LAN), a wide area network (WAN), or the Internet. In a preferred embodiment, computer **110** is a network computer. System **100** is suitable for use in the HotJava™ Views™ user environment, although one of skill in the art will recognize that methods and apparatus consistent with the present invention may be applied to other suitable user environments. HotJava Views is a graphical user interface developed by Sun Microsystems, Inc. for network computers. It is described, for example, in a document entitled Designing the HotJava Views™ User Environment For a Network Computer, Sun Microsystems, Inc. the contents of which are reproduced below.

Network Computers

The Network Computer (NC) represents a different approach to desktop computers. It is based on the Java Language and Web protocols, and has no permanent local storage. The operating system, applications, and data are centrally stored and loaded over the network as needed. Unlike terminals connected to a shared central computer, however, much of the application processing takes place on the local NC client, and therefore NCs do not have the scaling problems of X-Terminals where all the processing occurs on the central computer and bulky screen-display commands tend to clog the network. The arrival of the NC gives us a chance to rethink the basic design of computer applications.

The great promise of Network Computers is that they will reduce the burden of system administration and lower system support costs for large networks of client systems. Because all programs and data are centrally stored, updating software and backing up user files is a simple matter for the support staff. And if one of the NCs should have a hardware failure, it can simply be unplugged and replaced by a spare machine. Turning on the power switch is the only operation required to install new versions of the software on the client NC—the same procedure used every morning when these machines are turned on, because they do not have any permanent storage.

Another important advantage of NCs is that, because all the data is centrally stored on networked servers, it makes no difference which client machine a user logs in from. The user can access their data, programs, and normal environment from any machine that can connect to the network.

Although the central storage of programs and data has many important advantages, it also places serious constraints on NCs. In particular, all information on the client NC is lost every time the power is turned off, so the operating system, applications, and data must be downloaded from the network the first time they are needed after the power is turned on. Thus, an NC would probably not be a good choice for someone who needs a complex operating system for a variety of large applications, or large data files, such as a graphic designer.

User Environment for NCs

Sun Microsystems is producing an NC based on the Java language and JavaOS. The prototype machines were shipped with only the operating system and the HotJava Browser, with the intention that customers would create their business applications as Web-based applications running in the browser. But many potential customers also expressed an interest in a set of simple applications for employee communication and coordination. A group of human interface designers at Sun were asked to develop such a set of simple applications. That project has evolved into a product called HotJava Views, and the design of the human interface for HotJava Views is described below.

The HotJava Views Webtop user environment includes:

Selector—An intuitive "push-button" GUI

Mailview—Electronic mail

Calendar View—Electronic calendar

NameView—A directory of people in organization

InfoView—A Web browser for viewing intranet documents and (optionally) Internet documents

The Transaction Worker

The target users for HotJava Views are what we refer to as "transaction workers." Examples of transaction workers are order takers for a mail-order catalog business or customer support personnel at a bank's telephone call center. They use computers for a limited set of activities. Their primary-task applications were developed by an in-house MIS department or 3rd party developer, and typically access records in a central database. In addition to their primary tasks, transaction workers are also members of the organization, and need to communicate with other workers and management, attend meetings, and access company documents such as benefit plans and policy manuals. Currently these transaction workers are using 3270 terminals or PCS connected to a mainframe computer, but they have been identified by customer organizations as candidates for NCs, which can offer more functionality than terminals and simpler client administration than PCS.

Design Principles

The HotJava Views webtop user environment is based on a set of general design principles. Although we group these principles together in this paper, in fact they evolved in the course of designing the individual applications. Some principles were declared in advance; others were noticed only after they had been around for some time. As we recognized emerging principles during the design process, they were fed back to structure the overall design.

Switched Screens

Early on, we knew we would have a small, fixed set of applications for the first release.

We would provide mail, calendar, a directory of people, and a Web browser. There would also be a small set of applications for the user's primary tasks, developed in-house by the customer's MIS department. We needed a way for the user to access the applications and decided to simplify the model by letting the user view only one application at a time.

We were inspired by the 3270 terminal applications that our transaction workers often use now. These terminal applications usually have one function per screen and the users can quickly switch between screens when they want to access the different functions. By assigning each application (mail, calendar, etc.) to a separate screen and letting the use switch between screens, we could give the user easy access to the functions they needed without the clutter of overlapping applications windows. We considered a model based on tabbed cards, but tabs with text labels look best at the top of the screen and vertical space was at a greater premium than horizontal space on an NC monitor. Instead, we quickly settled on Selector, a switch containing a column of graphic icons along the left side, one for each screen. The user clicks on an icon to switch between screens. The state of a screen is saved when it is switched out and restored when it is switched back in. Because these applications are effectively always running, the user does not have to be concerned with starting and stopping applications.

The user switches between different application screens by clicking on one of the icons in the Selector along the left side of the screen. The current mail message contains a calendar appointment as an attachment. Clicking on the button under the pointer will schedule the appointment in the user's calendar.

Keep it Simple

Another early decision was to simplify the typical PC applications. We had several motivations. First, today's PC applications have generally become so loaded with features that many people use only a small portion of their functionality. In particular, our users would be focused primarily on their "transaction applications" and they would not need all the "power user" functions in applications such as electronic mail. Second, simpler applications would have reduced user training costs. Finally, to improve response time and reduce network traffic, the applications should be small enough to fit easily in the low memory size of an NC.

Our first attempt to simplify applications focused on electronic mail. We started with the mail program from the UNIX Common Desktop environment (CDE) and made a list of all its functions. We eliminated any function, preference or control that did not seem essential to the modest use of electronic mail that we expected from our target users, the transaction workers. We ended up throwing out about half of the functions in CDE mail and very few of those have crept back in as the design matured.

For example, we eliminated a whole set of functions for moving and copying messages to mailboxes, opening and closing mailboxes, and creating new mailboxes. Instead, because we expect transaction workers will be light users of email, we provided them with just a single folder for storing messages that they want to save and another folder where copies of outgoing messages are automatically saved. We also provided a set of commands for sorting and searching these mail folders, which could become moderately large. The net effect was that 21 mailbox-related commands in CDE mail were reduced to 7 commands in MailView. Similar reductions were made to other areas of functionality.

Our early decision to eliminate mail attachments was more problematical. Our users would not have access to a file system or to the typical PC applications needed to view attachments thus, we reasoned that mail attachments would be of limited use and they were excluded from early designs. However, feedback from customers, along with the realization that our users would be getting mail that might contain attachments from users of other email systems, led us to build in limited support for mail attachments. Our users can view and print some types of attachments in email from others, and they can also create new messages with calendar appointments and live URL links as attachments.

We are big fans of placing the most commonly used functions on a button bar where they are visible and can be more easily accessed than through the menus, but again we wanted a simple button bar, not a vast button cafeteria. So our next step was to try to fit our most important functions onto the button bar. To our surprise, we had done such a thorough job at reducing non-critical functions, that all the remaining functions fit on a single button bar. Buttons on the button bar are identified only by a graphic, but if the user's pointer lingers over a button for more than a fraction of a second, a text label appears below the button to describe its function.

No Menus

Having placed all the functions on the button bar, we were suddenly confronted with the unthinkable: If all the functions fit on the button bar, did we need menus at all? The pull-down menus of a typical GUI application arrange the application's functions into a hierarchy. Counting the menu names as one level, this hierarchy is often three and occasionally four levels deep. Menus allow the user to access a

large number of functions without permanently dedicating screen real estate for each of the functions. This advantage was especially important in computers with small screens. The negative aspect of menus is that the list of functions is hidden most of the time and it can be difficult or tedious to manipulate the menus. As typical computer screens have become larger, more and more of the menu functionality has been duplicated in tool bars, palettes, floating windows, and modeless dialogs, which keep the functions conveniently at hand. By eliminating menus and putting all the functionality directly in the main interface, we keep the functionality visible and easy to access.

Having ruthlessly simplified the mail application and eliminated menus, we were eager to know whether this approach would work with our other applications. When examining the CDE calendar, however, we did not find a large number of functions in the application menus that could easily be eliminated. So it was clear that the remaining menu functions would not fit into a reasonable button bar. Instead we took a different approach to simplification, and converted some of the menu operations into direct manipulation operations in the screen display. In the CDE calendar, the user is provided with both graphical and textual views of their weekly appointments, and they use a separate dialog to create, edit or delete an appointment. We chose to combine all the information into a single graphical view of the week's appointments that could be directly edited. In the CalenderView screen, new appointments are created by clicking at the desired time and typing in text for the appointment. The user can then change the times by dragging the top and bottom borders of an appointment. Appointments containing an envelope icon have an associated email message. Many of the functions that had been available from the menus were thus converted into direct manipulation of the appointment's graphical representation on the calendar.

Another technique that we used was to move functions from the main application menus to a place associated with their use. For example, both the CDE calendar and CalendarView provide a list of other people's calendars that you can also view. In CDE, both the list of other people and the facility for editing that list were in the main application menus. In HotJava Views we put the name of the calendar's owner at the upper left of the calendar and made it an option button, so that clicking on the button offers a choice of other people's calendars. The last item on the option list lets the user edit the list. This is another example of simplifying the interface by moving functionality from the main application menus to a place where it is more directly relevant.

The net result of these simplifications was that we were able to eliminate application menus from the calendar screen. In fact, there are no application menus in any of the HotJava Views applications.

Application Integration

Although our applications are simpler than typical PC applications, we wanted to improve the usability of the system by tightly integrating the applications so that they could easily be used together. At first we planned to use drag-and-drop as the primary technique for application integration. For example, an address from the NameView could be dragged and dropped onto the "To:" field of an email message. We soon realized, however, that there were several serious problems with drag-and-drop in our environment. First, how could we drag objects from one application to another, when only one application was displayed at a time? Second, many people find drag-and-drop difficult or slow, so in any event we would need to provide an alternative for keyboard-only operation. Such alternatives are normally made available from the menus, which we wanted to eliminate.

With drag-and-drop no longer a contender as the mechanism for application integration, we looked into the possibility of using buttons in the button bar. But what should happen when the user pressed such a button? We considered having the button switch to the other application, but that seemed disruptive. We also considered having the button bring up the other application in a window above the current application, but that violated our "rule" about having only one application visible at a time and led us down the path toward the standard overlapping window model. Finally, we developed a model in which a button in one application could display a "reduced version" of another application while keeping the focus centered in the current screen, and this turned out to be a very productive approach. We will explain the mechanism with an example.

Our user has received an email message announcing an upcoming meeting, and the message contains a calendar appointment as an attachment. The appointment attachment appears as an icon in the upper right comer of the message. When the user clicks the "Schedule appointment in Calendar" button in the button bar, a reduced version of the CalendarView application is displayed in a window in front of the mail screen. The reduced calendar shows the day of the appointment, with the new appointment scheduled and selected. The standard facilities for editing, printing, or deleting the appointment are available, but many of the other calendar functions are missing in this reduced application. For example, it is not possible to change to a month view or go to another day.

Although the window may appear to be similar to the standard overlapping windows of today's GUI interfaces, it is fundamentally different. The main application screen always remains as the background and cannot be resized. The reduced application window, the calender window, can be moved and resized, but always floats in front of the main application screen. The user can move the keyboard focus between the window and the screen, interact with the controls in either place, and even open additional reduced application windows, but the reduced application window always remains in front of the main application screen. Although the reduced application window is non-modal and can be moved out of the way, the user will typically treat it as a dialog. For example, when the user schedules an appointment from the MailView screen, the reduced calendar window is posted just for confirmation, and we expect the user to close the window almost immediately.

If the user later switches to the CalendarView application, they will see the new appointment with a small mail icon in the lower right comer of the appointment. The mail icon serves as a reminder that this appointment was scheduled from the mail application and provides another example of application integration. If the user wishes to see the original message, perhaps to get more information about the meting they can click on the mail icon and the original message is displayed in a reduced mail application window in front of the CalendarView.

Accessing Information

In addition to CalendarView and MailView, HotJava Views includes InfoView. InfoView is an HTML browser

intended primarily for viewing internal documents over the intranet. The Home Page button on InfoView will normally be configured to take the user to an index page for the organization's online manuals and documentation, and thus provide access to company information such as: policy, benefits, resources, and news.

The version of InfoView does not allow the user to type in arbitrary URLs, the user can only view pages that are accessible directly or indirectly from the Home Page or from one of the bookmarked pages. It is a simple matter, however, to configure InfoView so that it has a URL type-in field. As a further choice, customers also have the option of installing the standard HotJava Browser as an applet in the Selector in place of InfoView.

InfoView of course is Java enabled, so it can be used to present Web-based applications that contain Java applets. This facility provides a convenient way to give the user access to a variety of applications without putting them on the Selector. A button on the browser also allows the user to create a mail message that contains a live URL for the current page as an attachment.

There is also a reduced version of InfoView that appears in a window in conjunction with other applications. For example, it is used to view Web links included in a mail message, and to display the online Help.

Accessing People

NameView, provides an easy way to find people and related information. In addition to the usual name, address and phone number, NameView includes a network address for mail, calendar and the Web. Thus, after finding a person in NameView, a button click allows the user to send them mail, view their calendar, or browse their home page.

NameView is tightly integrated with the other HotJava Views applications. For example, a "reduced" NameView window can be used in MailView to address email messages, or in CalendarView to add someone to the list of calendars the user can browse.

Configuring the System

HotJava Views is highly configurable by the customer's system administrator. In addition to the applications described above, the Selector can be easily customized to contain one or more applications, such an order entry application, targeted at the user's primary job function. The Selector can expand to additional columns if needed to accommodate all the desired application icons.

The actual set of applications installed on the Selector, as well as many other aspects of the environment, can be specified for individual users or groups of users by the system administrator. And because Network Computers do not store data locally, the user is able to log in from any machine on the network and access their normal environment. The individual HotJava Views application can also be customized by the system administrator through the use of property files. For example, the default hours shown in the CalendarView screen are determined by the properties file, so that the initial calendar view can be set appropriately for workers on different shifts. Similarly, the properties file determines whether or not the InfoView application displays a field for entering arbitrary URLs.

Passing of the Old Order

Having given a flavor of the interface for our applications, we'll now describe the basic principles that we've followed and how they differ from the principles found in typical PC graphical user interfaces. First, we will discuss some common principles that we are not using.

The Desktop Metaphor

The desktop metaphor, with its overlapping windows, folders, documents and trash can, has been the basis for most graphical computer interfaces. Review of this article will show that the desktop metaphor has been banished. The static desktop has been replaced by active objects. Only one application screen is visible at a time. Views replace each other, rather than appearing as overlapping pieces of paper on a desktop. Although reduced application windows can overlap the main application screen and each other, they are temporary, second-class citizens in this world.

Our user never interacts with documents, folders, filing cabinets, or trash cans on the NC screen. One of our strategies for simplifying the interface is to minimize metaphors with objects in the physical world and develop models that are closely attuned to the computer-based activity. Basing the interface on a metaphor to a different realm always has a tendency to bring along irrelevant baggage.

Distinction Between Application and Document PC applications make a strong distinction between applications and documents, analogous to the computer science distinction between program and data. Users can separately manipulate applications and documents, for example with a file manager. The basic components of HotJava Views, however, are more like computer science "objects" with intertwined methods and data. Our applications act as viewers onto the data, and our users never interact with applications and documents separately. For example, they never see a CalendarView application that is not viewing someone's appointments, and they never encounter a calendar data file outside of the CalendarView application.

File System and File Manager

The lack of a distinction between applications and documents is closely related to the lack of a file system representation and a file manager in our interface. HotJava Views is essentially a set of information appliances that provide the only way to view data. The user deals with information objects, such as email messages or calendar appointments, rather than with files. Under the surface, these information objects may be represented as files or a database records on the central server, but that is of no concern to the user. The only file manager-like functionality that our users need is a way to choose among a relatively small, fixed set of application screens, and this is provided by the Selector on the left side of the screen. The installation and administration of the applications and data files is handled on the central server by a system administrator. The end user does not have to deal with file systems.

Double-Click to Open

The basic interaction model in PC graphical interfaces is to click on the icon representing an object to select it, and then choose an action on the selected object from a menu. Frequent use of menus can be a hassle and therefore a shortcut was added to the model: single-click to select, double-click to select and perform the default action. For most objects, such as documents, the default action is to open or view the object, and thus double-clicking on the object usually selects it and opens it.

In part because we do not distinguish between applications and documents, our users generally do not have a variety of operations that they could meaningfully perform on HotJava Views objects without first viewing them. Therefore we have yoked selection to viewing. Single-clicking an object selects it and displays it.

A New Approach

Our early decision to switch between single-application screens and our desire for simple, tightly integrated applications, led naturally to an interaction model that is somewhat different from the model for the typical PC application.

Single-Click, Single-Selection

Our basic model is that a single-click selects an object and displays it in the Selector, a single-click on one of the icons in the Selector on the left displays the corresponding object in the remainder of the screen. The selection is indicated by a rectangular border around the selected icon. Within the Active Mail folder, a message header is selected from a scrolling list and the body of the message is shown in the lower part of the display.

We allow only single selection from collections of objects. In most cases that is the only reasonable choice. Selecting more than one application in the Selector or more than one mailbox within the mail screen would not fit with our overall model of a single main application screen. In some cases, multiple selection might be reasonable. For example, the user might want to select several message headers and delete the messages with a single operation. We explored using multiple selection in these cases. However, in the interests of keeping the model simple and consistent, we have decided to restrict ourselves to single selection.

Web Look and Feel

You might have noticed that the graphic design in HotJava Views is somewhat reminiscent of the World Wide Web. This is intentional. We have tried to move away from the hardware look of contemporary graphical interfaces, in which the excessive use of borders and bevels has often resulted in a "boxy," visually noisy style. Instead, the Selector uses drop shadows without borders to achieve a floating three dimensional effect; we experimented both with shifting the graphics and reducing their shadows in order to achieve the "depressed" feel when the mouse button is down. Elsewhere, button bar graphics are angled and bleed to the edge of single-pixel borders. This downplays the repeating rectangle effect of bevel, white space, and icon border, and increases the space available for detail in the graphics without enlarging the size of the buttons. Throughout the interface, selected objects are marked by an enclosing border rather than a "depressed button" look.

The single-click model described earlier also fits with the feel of Web documents. We expect that our users will spend a significant portion of their time interacting with Web documents on the intranet. We didn't want to present them with one interaction style while they are using the browser and another interaction style while they are using other applications.

Button Integration

Our general interaction model is that the user selects an object to view it, and then has the option of clicking one of the buttons on the button bar for further operations on the selected object. Clicking a button to relate information to another application becomes a simple extension of this model. HotJava Views is written using, and manipulates the user interface via, the Java™ programming language. The Java programming language is described for example, in a text entitled "The Java Language Specification" by Janes Gosling, Bill Joy, and Guy Steele, Addison-Wesley, 1996, which is hereby incorporated by reference. Sun, Sun Microsystems, JavaSoft, the Sun Logo, Java, and HotJava Views are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

Network computer **110**, comprises several components that are all interconnected via a system bus **120**. Bus **120** is, for example, a bi-directional system bus connecting the components of network computer **110**, and contains thirty-two address lines for addressing a memory **125** and a thirty-two bit data bus for transferring data among the components. Alternatively, multiplex data/address lines may be used instead of separate data and address lines.

Network computer **110** contains a processor **115** connected to a memory **125**. Processor **115** may be microprocessor manufactured by Motorola, such as the 680X0 processor or a processor manufactured by Intel, such as the 80X86, or Pentium processor, or a SPARC™ microprocessor from Sun Microsystems, Inc. However, any other suitable microprocessor or micro-, mini-, or mainframe computer, may be utilized. Memory **125** may include a RAM, a ROM, a video memory, and mass storage. The mass storage may include both fixed and removable media (e.g., magnetic, optical, or magnetic optical storage systems or other available mass storage technology).

The user typically inputs information to network computer **110** via a keyboard **130** and a pointing device, such as a mouse **135**, although other input devices may be used. In return, information is conveyed to the user via display screen **140**.

Computer **110** communicates with other computers on network **170** via a network interface **145**, examples of which include Ethernet or dial-up telephone connections. Accordingly, computer **110** sends messages and receives data, including program code, through network **170**. In accordance with the invention, one such downloaded task is the user environment application program described herein. The received code may be executed by processor **115** as it is received, and/or stored in memory **125** for later execution. Application code may be embodied in any form of computer program medium. The computer systems described herein are for purposes of example only. An embodiment of the invention may be implemented in any type of computer system or programming or processing environment.

The operating system, tasks, and data required by network computer **110** are centrally stored at server **180**, and are loaded over network **170** as needed. When network computer **110** is first turned on, server **180** provides it with a boot image that includes the operating system, such as JavaOS 1.0, developed by Sun Microsystems, Inc. Server **180** also provides to network computer **110** the HotJava Views user interface. This user interface supports multiple workspaces, such as, for example: an electronic mail workspace, an electronic calendar workspace, a workspace containing a directory of people in the organization, and a web browser workspace for viewing documents on an intranet or on the Internet. The actual set of tasks installed, and correspondingly, the workspaces available, can vary depending on how the system is configured by a systems administrator.

FIG. **2***a* is a representative user interface screen showing a workspace consistent with the present invention. FIG. **2***a* shows a selector **210**, which is a vertical column on the left side of the screen that contains an icon corresponding to each workspace. Selector **210** functions as a workspace switch that allows the user to switch between the different workspaces. The user selects a workspace by clicking on one of the icons in selector **210**. In response, computer **110** displays the windows contained in that workspace.

Selector **210** is created by a user environment (UE) application program that is part of HotJava Views and is provided to network computer **110** by server **180** during startup. The UE application program is responsible for creating and managing the graphical user interface used in HotJava Views. For example, the UE application program maintains in memory **125** an internal data structure that records which windows have been opened in which workspaces, along with their current location and size.

In FIG. **2***a*, selector **210** contains icons corresponding to multiple workspaces including Mail, Personal Organization,

and other workspaces such as Workspace N. The Personal Organization workspace is active, as represented by the bold box surrounding its icon in selector **210**, and is displayed in screen area **225**. The Personal Organization workspace contains two windows: one corresponding to a Notepad task (which may allow the user to create and store short messages), and one corresponding to a Schedule task (which may show activities for a particular day, week or month).

The user interface also contains a sliding window panel **220** that contains an icon for every window in every workspace, whether visible or hidden. As a default, this icon list is presented vertically in alphabetical order, but the user may rearrange the ordering of the icon list if desired. As shown in FIG. **2***a*, window panel **220** contains icons for three windows: Inbox, Notepad, and Schedule. Window panel **220** contains icons corresponding to all three windows, even though only two windows are displayed in the selected workspace.

In one embodiment, window panel **220** is placed adjacent to selector **210**. To minimize cluttering the user interface, window panel **220** is normally hidden. When hidden, its existence is indicated to the user by a thin, vertical strip along the left edge of screen **140**. The user can open window panel **220** by moving the pointer to the left edge of screen **140**. This causes window panel **220** to slide out from under selector **210** into the position shown in FIG. **2***a*.

FIG. **2***b* is a representative user interface screen showing another workspace consistent with the present invention; the screen shown in FIG. **2***b* corresponds to the screen shown in FIG. **2***a*. In FIG. **2***b*, the Mail workspace is active, as represented by the bold box surrounding its icon in selector **210**, and is displayed in screen area **225**. The Mail workspace contains two windows: one corresponding to an Inbox task (which may show incoming electronic mail messages), and one corresponding to the Notepad task shown in FIG. **2***a*. Again, window panel **220** contains icons corresponding to all three windows, even though only two windows are displayed in the selected workspace.

FIG. **2***b* also shows a drawer **230**, which contains icons corresponding to tasks that can be launched. Drawer **230** contains icons corresponding to tasks such as Notepad, Spreadsheet and others such as Task X.

C. Architectural Operation

For purposes of explanation, the following description is based on the screen displays shown in FIGS. **2***a* and **2***b*. Those of skill in the art will recognize that the combinations shown herein are exemplary only.

FIG. **3** is a flow diagram of operations performed to launch the Notepad task shown in FIG. **2***b* in a manner consistent with the present invention. The process begins when system **100** receives a command from the user selecting the Mail workspace (step **310**) by clicking on its icon in selector **210**. After the Mail workspace has been selected, system **100** receives a command from the user to launch the Notepad task (step **320**). In one embodiment, the user does so by using mouse **135** to move the pointer to the bottom of the screen; this opens drawer **230**, which contains icons for various tasks that may be launched. The user points to the icon corresponding to Notepad and clicks the mouse button to launch the task.

In response to the user's command to launch the Notepad task, processor **115** checks to determine whether the Notepad task is already displayed in a window in the Mail workspace (step **330**). If so (as in this example), processor **115** causes the corresponding window to be brought to the foreground of the display (step **360**), and the procedure ends. If the Notepad task had not already been displayed in a window in the Mail workspace, processor **115** would cause a window containing the Notepad task to be created and shown on display screen **140** (step **340**). In addition, processor **115** would cause an icon to be added to window panel **220**, corresponding to the Notepad task that was just launched (step **350**).

FIG. **4** is a flow diagram of operations performed to hide the Notepad window in the Mail workspace, and assumes that the Notepad window is displayed in both the Mail and Personal Organization workspaces. The procedure begins when system **100** receives a command from the user selecting the Mail workspace, which may be done by using selector **210** (step **410**). In response, processor **115** displays on screen **140** all windows shown in that workspace. System **100** then receives a command from the user that indicates that the user wishes to hide the Notepad window (step **420**); this may be accomplished by having the user click on a designated button in the frame of the Notepad window. In response to this command, processor **115** stores the size and position of the Notepad window in memory **125** (step **430**). This is done so that if the user subsequently desires to view the window, it can be displayed at the same position and at the same size as it previously existed in the workspace. Finally, processor **115** causes the Notepad window to be removed from the display of the Mail workspace (step **440**).

Importantly, although the window is hidden in the Mail workspace, any corresponding version of the window that is open in another workspace (such as in the Personal Organization workspace shown in FIG. **2***a*) remains visible in that workspace. Moreover, although the Notepad window is hidden in the current workspace, the icon representing that window remains in window panel **220**. This is necessary so that the user can later show the Notepad window in the Mail workspace by using window panel **220**.

FIG. **5** is a flow diagram of operations performed to show a hidden or obscured window, and assumes that the Notepad window is hidden in both the Mail and Personal Organization workspaces. The procedure begins when system **100** receives a command from the user selecting a workspace, which may be done by using selector **210** (step **505**). For example, the user may select the Mail workspace. System **100** then receives a command from the user to access window panel **220**, which may be accomplished by the user pointing to the left edge of screen **140** (step **510**). In response to the user's pointing, window panel **220** slides open next to selector **210** (step **515**).

Once window panel **220** is displayed, system **100** receives a command from the user to display the window; the user may issue this command by using mouse **135** to click on an icon representing the Notepad window (step **520**). In response, processor **115** first checks to determine whether the Notepad window is already being displayed in the current workspace (step **525**). If it is, processor **115** causes the Notepad window to be moved to the foreground of display **140** (step **530**). This allows the user to bring to the foreground a window that already exists in the current workspace, but has been obscured or hidden by other windows.

If the selected window is not already displayed in the current workspace (as in this example), processor **115** checks to determine whether the selected window has previously been displayed in the current workspace (step **535**). If so, processor **115** recalls from memory **125** the size and position of the window as it was previously displayed (step **540**). If not, a default size and position are used for the window. Next, processor **115** causes the Notepad window to be displayed in the Mail workspace (step **545**), using either

stored or default parameters. Importantly, although the window may be displayed in the Mail workspace, it does not affect the display (or lack thereof) of a corresponding window in another workspace, such as the Personal Organization workspace.

FIG. **6** is a flow diagram of operations performed to terminate the Notepad window from the Personal Organization workspace, and assumes that the Notepad window is displayed in both the Mail and Personal Organization workspaces. The procedure begins when system **100** receives a command from the user selecting the Personal Organization workspace (step **610**). If the Personal Organization workspace did not contain the window, an appropriate workspace could be selected by using selector **210**. System **100** then receives a command from the user to terminate the Notepad window (step **620**); the user may issue this command by clicking on a designated button on the Notepad window. Alternatively, the user may terminate a window directly from window panel **220** by pointing at the icon representing that window and using a pop-up menu that contains a "terminate" command. In response, processor **115** removes the Notepad window from the display of all workspaces (step **630**). Processor **115** also causes the icon representing the Notepad window to be removed from window panel **220** (step **640**), and the procedure ends.

FIG. **7** is a flow diagram of steps performed to reposition or resize the Notepad window in the Mail workspace, and assumes that the Notepad window is displayed in both the Mail and Personal Organization workspaces. The procedure begins when system **100** receives a command from the user selecting the Mail workspace, which may be done by using selector **210** (step **710**). In response, processor **115** causes to be displayed on screen **140** all windows contained in that workspace. System **100** then receives a command from the user selecting the Notepad window for repositioning or resizing (step **720**), which may be done by using mouse **135**. System **100** then receives a command from the user to reposition or resize the Notepad window (step **730**), again by using mouse **135**. In response to the user's actions, processor **115** causes the Notepad window's size or position to change on display screen **140** (step **740**), and the procedure ends. Importantly, the user's actions in repositioning or resizing the Notepad window in the Mail workspace does not affect the position or size of the corresponding Notepad window in other workspaces, such as the Personal Organization workspace.

D. Conclusion

As described in detail above, methods and apparatus consistent with the present invention allow a user to easily select and manipulate windows in a user interface that supports multiple workspaces. The foregoing description of an implementation of the invention has been presented for purposes of illustration and description. For example, the described implementation includes software but the present invention may be implemented as a combination of hardware and software or in hardware alone. Modifications and variations are possible to the processes described in connection with FIGS. **3–7** in light of the above teachings or may be acquired from practicing the invention.

Although systems and methods consistent with the present invention are described as operating in the exemplary distributed system and the Java™ programming environment, one skilled in the art will appreciate that the present invention can be practiced in other systems and programming environments. Additionally, although aspects of the present invention are described as being stored in memory, one skilled in the art will appreciate that these aspects can also be stored on other types of computer-readable media, such as secondary storage devices, like hard disks, floppy disks, or CD-ROM; a carrier wave from the Internet; or other forms of RAM or ROM. The scope of the invention is therefore defined by the claims and their equivalents.

What is claimed is:

**1**. A method for managing the display of windows corresponding to tasks executable by a computer, the method comprising:

providing multiple workspaces, each workspace being capable of displaying multiple windows corresponding to executing tasks;

displaying a window panel including icons corresponding to the executing tasks; and

permitting shared access to a window of the executing tasks upon selection of a corresponding icon displayed in the window panel.

**2**. The method of claim **1**, further comprising adding to the window panel an icon for each task opened into a window.

**3**. The method of claim **1**, further comprising:

receiving a command from a user to launch a task;creating a window for the task; and

adding an icon representing the window to the window panel.

**4**. The method of claim **1**, further comprising:

receiving a command from a user to hide a window in a selected workspace;

removing the window only from the selected workspace; and

maintaining an icon representing the window in the window panel.

**5**. The method of claim **4**, further comprising recording a size and location for the window in the selected workspace.

**6**. The method of claim **1**, further comprising:

receiving a command from a user, in a particular workspace, selecting an icon in the window panel; and

displaying in the foreground of the workspace a window corresponding to the selected icon.

**7**. The method of claim **1**, further comprising the steps of:

receiving a command from a user to terminate a window;

removing the window from each workspace in which it is displayed; and

deleting from the window panel an icon corresponding to the terminated window.

**8**. The method of claim **1**, further comprising the steps of:

displaying a window in more than one workspace; and

resizing the window in one workspace without affecting the size of the window in another workspace.

**9**. The method of claim **1**, further comprising the steps of:

displaying a window in more than one workspace; and

repositioning the window in one workspace without affecting the position of the window in another workspace.

**10**. The method of claim **1**, further comprising the steps of:

hiding the window panel behind a workspace switch;

receiving a command from the user selecting the workspace switch; and

displaying the window panel in response to the user command selecting the switch.

11. A computer-readable medium containing instructions for managing the display of windows corresponding to tasks executable by a computer, by:

providing multiple workspaces, each workspace being capable of displaying multiple windows corresponding to executing tasks;

displaying a window panel including icons corresponding to the executing tasks; and

permitting shared access to a window of the executing tasks upon selection of a corresponding icon displayed in the window panel.

12. The computer-readable medium of claim 11, further comprising adding to the window panel an icon for each task opened into a window.

13. The computer-readable medium of claim 11, further comprising:

receiving a command from a user to launch a task;

creating a window for the task; and

adding an icon representing the window to the window panel.

14. The computer-readable medium of claim 11, further comprising:

receiving a command from a user to hide a window in a selected workspace;

removing the window only from the selected workspace; and

maintaining an icon representing the window in the window panel.

15. The computer-readable medium of claim 14, further comprising recording a size and location for the window in the selected workspace.

16. The computer-readable medium of claim 11, further comprising:

receiving a command from a user, in a particular workspace, selecting an icon in the window panel; and

displaying in the foreground of the workspace a window corresponding to the selected icon.

17. The computer-readable medium of claim 11, further comprising the steps of:

receiving a command from a user to terminate a window;

removing the window from each workspace in which it is displayed; and

deleting from the window panel an icon corresponding to the terminated window.

18. The computer-readable medium of claim 11, further comprising the steps of:

displaying a window in more than one workspace; and

resizing the window in one workspace without affecting the size of the window in another workspace.

19. The computer-readable medium of claim 11, further comprising the steps of:

displaying a window in more than one workspace; and

repositioning the window in one workspace without affecting the position of the window in another workspace.

20. The computer-readable medium of claim 11, further comprising the steps of:

hiding the window panel behind a workspace switch;

receiving a command from the user selecting the workspace switch; and

displaying the window panel in response to the user command selecting the switch.

21. An apparatus for managing the display of windows corresponding to tasks executable by a computer, comprising:

a memory having program instructions, and

a processor configured to use the program instructions to:

provide multiple workspaces, each workspace being capable of displaying multiple windows corresponding to executing tasks;

display a window panel including icons corresponding to the executing tasks; and

permit shared access to a window of the executing tasks upon selection of a corresponding icon displayed in the window panel.

22. The apparatus of claim 21, wherein the processor is further configured to add to the window panel an icon for each task opened into a window.

23. The apparatus of claim 21, wherein the processor is further configured to:

receive a command from a user to launch a task;

create a window for the task; and

add an icon representing the window to the window panel.

24. The apparatus of claim 21, wherein the processor is further configured to:

receive a command from a user to hide a window in a selected workspace;

remove the window only from the selected workspace; and

maintain an icon representing the window in the window panel.

25. The apparatus of claim 24, wherein the processor is further configured to record a size and location for the window in the selected workspace.

26. The apparatus of claim 21, wherein the processor is further configured to:

receive a command from a user, in a particular workspace,

selecting an icon in the window panel; and

display in the foreground of the workspace a window corresponding to the selected icon.

27. The apparatus of claim 21, wherein the processor is further configured to:

receive a command from a user to terminate a window;

remove the window from each workspace in which it is displayed; and

delete from the window panel an icon corresponding to the terminated window.

28. The apparatus of claim 21, wherein the processor is further configured to:

display a window in more than one workspace; and

resize the window in one workspace without affecting the size of the window in another workspace.

29. The apparatus of claim 21, wherein the processor is further configured to:

display a window in more than one workspace; and

reposition the window in one workspace without affecting the position of the window in another workspace.

30. The apparatus of claim 21, wherein the processor is further configured to:

hide the window panel behind a workspace switch;

receive a command from the user selecting the workspace switch; and

display the window panel in response to the user command selecting the switch.

31. An apparatus for managing the display of windows corresponding to tasks executable by a computer, comprising:

means for providing multiple workspaces, each workspace being capable of displaying multiple windows corresponding to executing tasks;

means for displaying a window panel including icons corresponding to the executing tasks; and

means for permitting shared access to a window of the executing tasks upon selection of a corresponding icon displayed in the window panel.

32. The apparatus of claim 31, further comprising means for adding to the window panel an icon for each task opened into a window.

33. The apparatus of claim 31, further comprising:

means for receiving a command from a user to launch a task;

means for creating a window for the task; and

means for adding an icon representing the window to the window panel.

34. A method for managing the display of windows corresponding to tasks executable by a computer, the method comprising:

providing multiple workspaces, each workspace being capable of displaying multiple windows corresponding to executing tasks;

displaying a first window panel including first icons corresponding to executing workspaces;

displaying a first workspace, the first workspace displaying at least one first workspace window corresponding to an executing task associated with at least the first workspace; and

displaying, simultaneously with the first window panel, said first workspace and said first workspace window, a second window panel including second icons corresponding to said executing tasks.

35. The method of claim 34, further including the steps of:

receiving a command from a user selecting a particular second icon in the second window panel, the particular second icon reflecting a task executed in both said first and second workspaces; and

displaying in the foreground of the first workspace, a window corresponding to the task reflected by the particular second icon.

36. The method of claim 34, wherein displaying a first workspace includes the step of:

receiving a command from a user selecting a particular first icon in the first window panel, said particular first icon corresponding to said first workspace.

37. The method of claim 35, further including the steps of:

displaying a third window panel including icons corresponding to non-executing tasks and icons corresponding to said executing tasks;

receiving a second command from a user selecting an icon corresponding to a first non-executing task;

initiating execution of the first non-executing task;

generating a window corresponding to the first non-executing task, the generated window representing the execution of the first non-executing task; and

displaying the generated window in said first workspace.

38. A system for managing the display of windows corresponding to tasks executable by a computer, the system comprising:

means for providing multiple workspaces, each workspace being capable of displaying multiple windows corresponding to executing tasks;

means for displaying a first window panel including first icons corresponding to executing workspaces;

means for displaying a first workspace,

means for displaying within the first workspace at least one first workspace window corresponding to an executing task associated with at least the first workspace; and

means for displaying, simultaneously with the first window panel, said first workspace and said first workspace window, a second window panel including second icons corresponding to said executing tasks.

39. The system of claim 38, further comprising:

means for receiving a command from a user selecting a particular second icon in the second window panel, the particular second icon reflecting a task executed in both said first and second workspaces; and

means for displaying in the foreground of the first workspace, a window corresponding to the task reflected by the particular second icon.

40. The method of claim 38, wherein the means for displaying a first workspace further comprises:

means for receiving a command from a user selecting a particular first icon in the first window panel, said particular first icon corresponding to said first workspace.

41. The system of claim 39, further comprising:

means for displaying a third window panel including icons corresponding to non-executing tasks and icons corresponding to said executing tasks;

means for receiving a second command from a user selecting an icon corresponding to a first non-executing task;

means for initiating execution of the first non-executing task;

means for generating a window corresponding to the first non-executing task, the generated window representing the execution of the first non-executing task; and

means for displaying the generated window in said first workspace.

42. A method for managing the display of windows corresponding to tasks executable by a computer, the method comprising:

providing multiple workspaces, each workspace capable of displaying multiple windows corresponding to multiple executing tasks;

providing a first window panel including icons representing executing workspaces;

providing a second window panel including icons representing executing tasks corresponding to the executing workspaces;

displaying in a display area, the first window panel, second window panel, an active workspace, and at least one window representing an executing task within the active workspace;

receiving a command from a user to launch a task;

creating a new window for the task; and

adding an icon representing the created window to the second window panel.

43. The method of claim 42, wherein the first window panel includes a highlighted icon representing the active workspace displayed in the display area, and the second window panel includes one or more icons representing executing tasks, including executing tasks not associated with the active workspace.

44. The method of claim 43, wherein the second window panel is a sliding window panel.

45. The method of claim 43, wherein the display area includes a third window panel including icons representing non-executing tasks.

46. The method of claim 45, including the steps:

selecting a third window panel icon representing a first non-executing task;

initiating execution of the first non-executing task represented by the third window panel icon, wherein the first non-executing task is transformed into a first executing task;

creating a new window in the active workspace, the new window representing the first executing task; and

creating a first executing task icon in the second window panel.

47. The method of claim 43, including the steps:

receiving a command to terminate an executing task represented by an executing task window in the first active workspace;

terminating the execution of the executing task;

removing the executing task window from the active workspace; and

removing an icon representing the terminated executing task, from the second window panel.

* * * * *